VocalFusion® XVF3510 VOICE PROCESSOR

USER GUIDE

V4.2

Bringing technology to life

CONTENTS

XMOS
Bringing technology to life

1. XVF3510 USER GUIDE

1.1. SCOPE OF DOCUMENT

The XMOS VocalFusion® XVF3510 User Guide is written for system architects and engineers designing Far-field voice systems using the XVF3510 voice processor. The document describes typical usage models, the processor architecture, key feature operation, and interface definitions. In conjunction with the product datasheet, these two documents provide all the information required for system design, from concept to production testing and verification.

It is expected that this document is read in conjunction with the relevant datasheet and that the user is familiar with basic voice processing terminology.

NOTE: This issue of the user guide covers the functionality supported by version 4.2 of the VocalFusion XVF3510 application firmware.

1.2. XVF3510 FAR-FIELD VOICE PROCESSORS

The XMOS XVF3510 range of voice processors uses microphone array processing to capture clear, high-quality voice from anywhere in the room. XVF3510 processors use highly optimised digital signal processing algorithms implementing 'barge-in', point noise and ambient noise reduction to increase the Signal-to-Noise Ratio (SNR) achieving a reliable voice interface whatever the environment.

The XVF3510 processor is designed for seamless integration into consumer electronic products requiring voice interfaces for Automatic Speech Recognition (ASR), communications or conferencing. In addition to its class-leading voice processing, the XVF3510 voice processor provides a comprehensive set of interfaces and configuration options to simplify the integration of a voice interface into a wide range of system architectures. This includes specific features required in TV and set-top box applications, including audio switching and digital inputs and outputs that support switches and LED indicators.

The XVF3510 voice processor executes a firmware image that is either read from a flash memory device or loaded by a host processor. The Device Firmware Upgrade (DFU) function of the processor allows in field upgrade ensuring all products can benefit from the latest releases. While the voice processor is running, this configuration can be modified by the host system over the XVF3510 control interface. The control interface also allows the host system to control peripheral devices and obtain status information from the device and its digital inputs.

Two variants of the XVF3510 are available which have been optimised for different application use cases. These two variants require different firmware to be loaded onto the device.

Table 1-1 XVF3510 variants

PRODUCT	KEY FEATURES	TARGET APPLICATION
XVF3510-INT	Far-field voice interface Audio interface: I2S (Slave) Control interface: I2C (Slave) Device Firmware Upgrade: I2C (Slave)	Voice interface integrated into the product
XVF3510-UA	Far-field voice interface Audio interfaces: USB UAC1.0 (and optionally I2S Master) Control interface: USB2.0 Full Speed Device Firmware Upgrade: USB	USB plug-in voice accessory, and integrated products using USB

These application use cases are described in more detail in the following sections.

1.3. SYSTEM BLOCK DIAGRAMS

1.3.1. XVF3510-INT CONFIGURATION

The XVF3510-INT device has been optimized for integration on a system board. A standard I2C interface is provided to enable the main processor on the system board to configure and monitor the XVF3510-INT. The processed voice signal is output over an I2S bus to the host system and the XVF3510 receives its I2S audio reference signal for the Acoustic Echo Cancellation function.

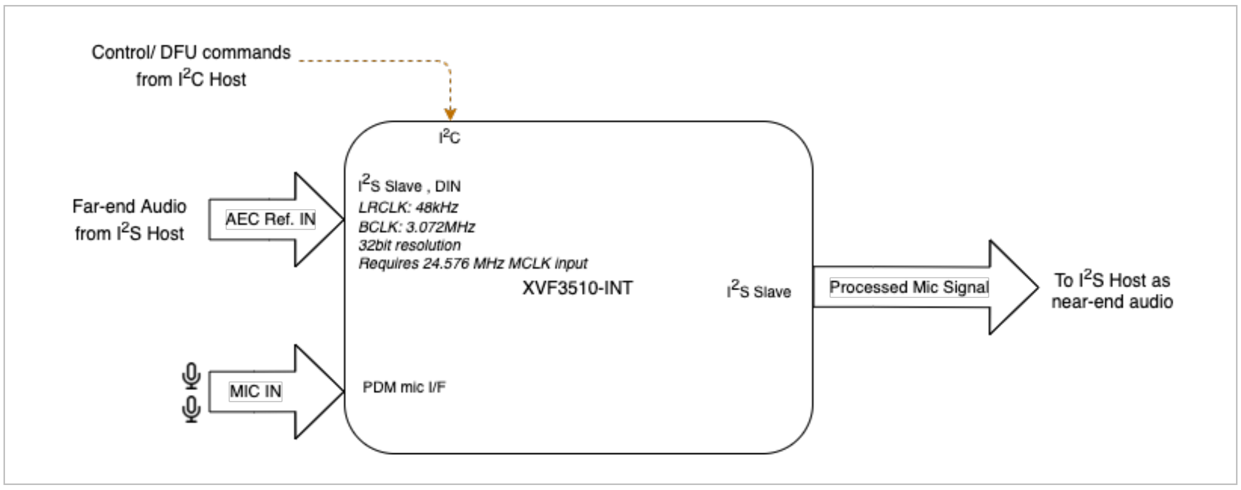


Figure 1-1 XVF3510-INT Integrated configuration

1.3.2. XVF3510-UA CONFIGURATION

The XVF3510-UA device replaces the I2C interface of the XVF3510-INT with a USB2.0 compliant PHY which supports a UAC1.0 audio device for both reference signal input and processed audio output. In addition, the USB device supports a standard USB Endpoint 0 for device control and a standard USB HID for status events. An optional I2S master interface is also available on the device to output an audio signal to an external audio device.

The following block diagram illustrates the typical configuration.

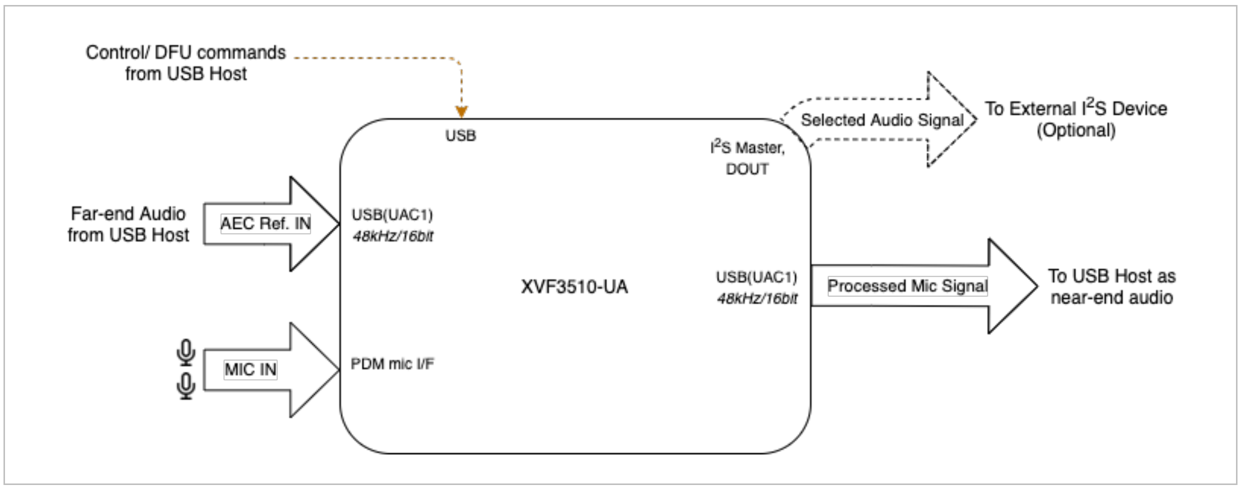


Figure 1-2 XVF3510-UA Configuration for USB-only use case

In addition to the standard USB configuration shown above, the XVF3510-UA also supports an alternative configuration in which the AEC reference signal is supplied over an I2S bus.

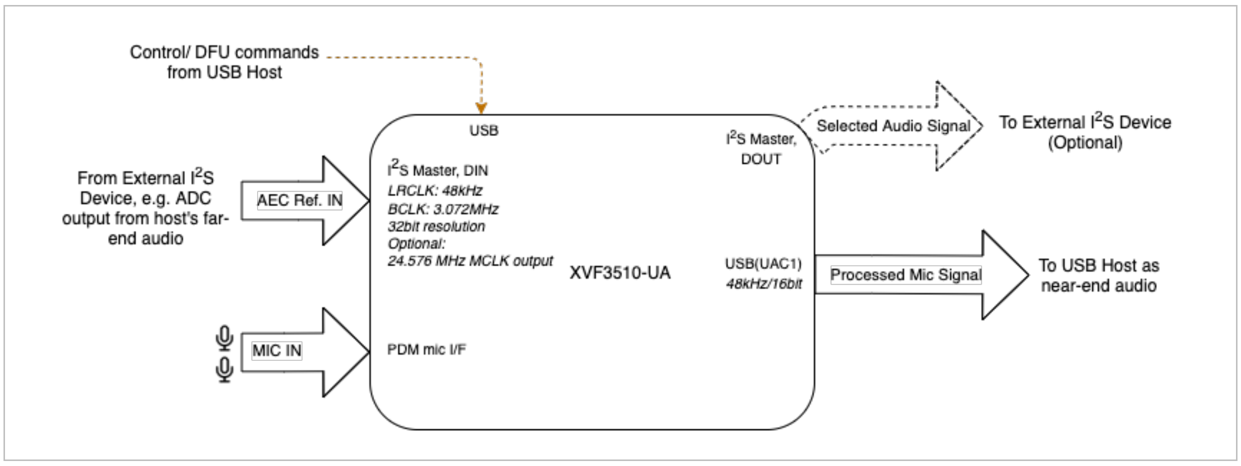


Figure 1-3 XVF3510-UA Configuration using I2S audio reference

1.4. DEVICE FIRMWARE AND CONFIGURATION

The operation of the XVF3510 device is controlled through a firmware image that is loaded onto the device when it is powered up. Two modes of operation are supported:

- The firmware image can either be stored in a QSPI Flash device which is read by the XFV3510 processor automatically, or
- The firmware image is downloaded to the XVF3510 processor over the SPI interface by the host processor on the system board.

Selection of the boot mode is made via setting the QSPI_D1/BOOTSEL pin on the device as described in the datasheet.

The firmware image configures the XVF3510 into a standard, default operational mode. This mode can be modified at startup via a set of configuration parameters that are stored in the flash device along with the firmware in the XVF3510 Data Partition. These commands can be used to reconfigure the device during startup, and also initialise other devices attached to it.

If the device firmware is downloaded from the host, then the data partition is not required and the device is configured directly over the control interface.

2. XVF3510 VOICE PROCESSOR ARCHITECTURE

2.1. OVERVIEW

The core of the XVF3510 voice processor is a high-performance audio processing pipeline that takes its input from a pair of the microphone and executes a series of signal processing algorithms to extract a voice signal from a complex soundscape. The audio pipeline can accept a reference signal from a host system which is used to perform Acoustic Echo Cancellation (AEC) to remove audio being played by the host. The audio pipeline provides two different output channels - one that is optimized for Automatic Speech Recognition systems and the other for voice communications.

Flexible audio signal routing infrastructure and a range of digital inputs and outputs enable the XVF3510 to be integrated into a wide range of system configurations, that can be configured at start up and during operation through a set of control registers.

In addition, the XVF3510-UA variant supports a standard USB PHY interface which supports a UAC audio device and device control over USB. The following sections describe the voice pipeline and the surrounding infrastructure in more detail.

2.2. AUDIO PROCESSING PIPELINE

The audio processing pipeline is common to both the XVF3510-UA and XVF3510-INT firmware variants. The signal processing chain is described below, with individual blocks and usage described in more detail in subsequent sections.

The XVF3510 audio processing pipeline takes inputs from a pair of MEMS Pulse Density Modulation (PDM) microphones and uses advanced signal processing to create audio streams suitable for use in Automatic Speech Recognition (ASR) and voice communication applications. The pipeline enhances the captured audio stream using a set of complementary signal enhancement and noise reduction processes.

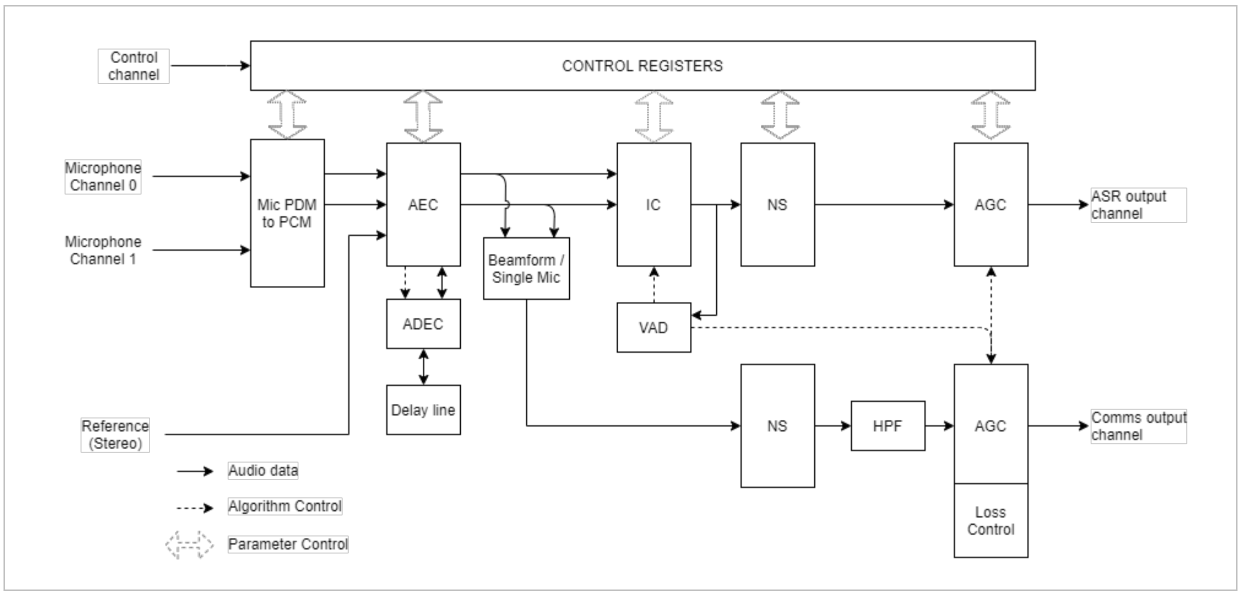


Figure 2-1 The XVF3510 audio pipeline

The pipeline takes its input from a pair of low-cost PDM microphones and converts this signal to PCM for further processing:

- **Acoustic Echo Cancellation (AEC):** Continuously modelling the room acoustics allows the AEC to remove audio being played into the room by the product which the XVF3510 is a component of. A reference copy of the audio is provided to the AEC in order for it to accurately estimate the echo.
- **Automatic Delay Estimation & Control (ADEC):** Automatically monitors and automatically compensates for the delay between the reference audio and the echo received by the microphone.

Following echo cancellation, the ASR and communications paths diverge to permit parameter tuning appropriate for the individual audio output use cases.

- **Interference Cancellation (IC):** Suppresses static noise from point sources such as cooker hoods, washing machines, or radios for which there is no reference audio signal available.
- **Voice Activity Detection (VAD):** Controls adaption the IC and AGC to optimise output for near-end speech.
- **Noise Suppression (NS):** Suppresses diffuse noise from sources whose frequency characteristics do not change rapidly over time (i.e., diffuse stationary noise).
- **Automatic Gain Control (AGC):** Controls the audio output level via separate AGC channels for Automatic Speech Recognition (ASR) and communications output. The VAD is used to prevent gain changes during speech to improve speech recognition performance.

The pipeline has been designed to minimise the need to tune and modify these functions. However, if required for specific use cases, these later sections of this document provide details of the relevant parameters and processes.

2.3. ASR AND COMMUNICATION PROCESSING

The audio pipeline discussed above produces two separate audio streams, one specifically tuned for integration with keyword and ASR services and the other designed for conferencing and communication applications. Both processed audio streams are available simultaneously on the left and right channels of the USB and I2S audio outputs. The default configuration is as follows:

Table 2-1 Default channel mapping (both USB and I2S)

CHANNEL	DEFAULT
[0] - Left	ASR
[1] - Right	Communications

In situations where an ASR is used to invoke a call it may be necessary to continually monitor the ASR channel for a ‘end call’ intent. The parallel output of both ASR and Communications processed streams allow the combination of high-quality calling audio with the tuned ASR capability.

The IO_MAP configuration parameter (see *Signal flow and processing* section) allows users to also configure both channels to be ASR or Communications if required.

2.4. XVF3510-INT - FOR INTEGRATED VOICE INTERFACE APPLICATIONS

The XVF3510-INT product embeds the core audio processing pipeline in an audio infrastructure that supports rate conversion, filtering and signal routing. This infrastructure is controllable by the host system via a set of control registers. In addition, the XVF3510-INT provides a set of peripheral interfaces to the host system to other devices, eg digital inputs, LEDs, SPI peripherals etc.

The peripheral interfaces supported include an interface to an optional QSPI Flash device containing the XVF3510 firmware and configuration information that is loaded by the processor on startup.

The system architecture of the XVF3510-INT is shown below.

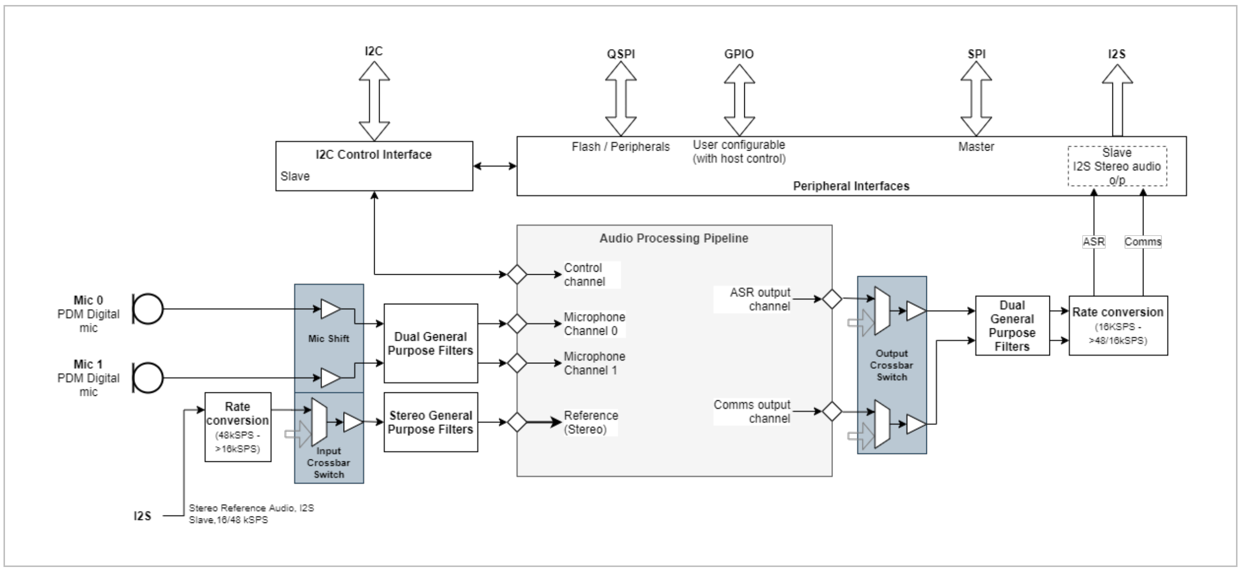
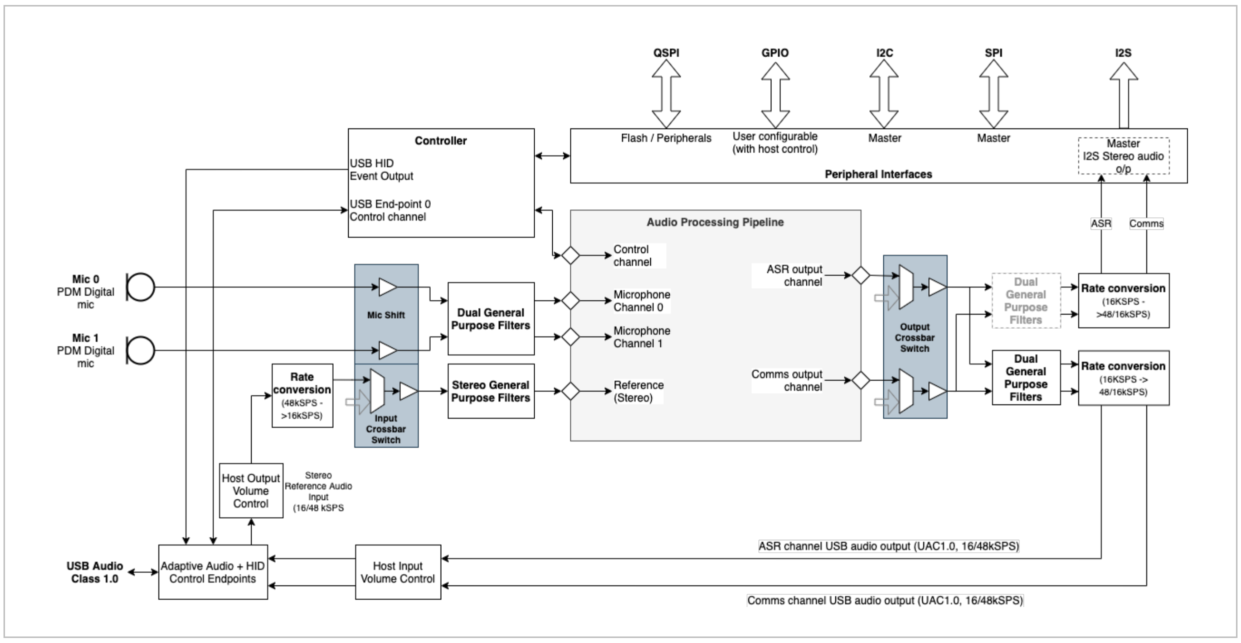


Figure 2-2 XVF3510-INT System architecture

2.5. XVF3510-UA - FOR USB ACCESSORY VOICE INTERFACE APPLICATIONS

The XVF3510-UA variant includes the same audio infrastructure as the XFV3510-INT, but it includes a USB interface that implements a UAC1.0 audio device to interface to the host system. The USB interface also supports an Endpoint 0 control channel, and a USB HID to signal input events to the host.

The system architecture of the XVF3510-UA is shown below.



QSPI
GPIO
I2C
SPI
I2S
Controller
USB HID Event Output
USB End-point 0 Control channel
Flash / Peripherals
User configurable (with host control)
Master
Master
Master I2S Stereo audio o/p
Peripheral Interfaces
ASR
Comms
Audio Processing Pipeline
Control channel
ASR output channel
Mic 0 PDM Digital mic
Mic 1 PDM Digital mic
Mic Shift
Dual General Purpose Filters
Microphone Channel 0
Microphone Channel 1
Output Crossbar Switch
Dual General Purpose Filters
Rate conversion (16KSPS - >48/16KSPS)
Dual General Purpose Filters
Rate conversion (16KSPS -> 48/16KSPS)
Rate conversion (48kSPS - >16kSPS)
Input Crossbar Switch
Stereo General Purpose Filters
Reference (Stereo)
Comms output channel
Host Output Volume Control
Stereo Reference Audio Input (16/48 kSPS
USB Audio Class 1.0
Adaptive Audio + HID Control Endpoints
Host Input Volume Control
ASR channel USB audio output (UAC1.0, 16/48kSPS)
Comms channel USB audio output (UAC1.0, 16/48kSPS)

Figure 2-3 XVF3510-UA System architecture

NOTE: The XVF3510-UA product also supports a hybrid mode of operation where the reference signal is delivered via I2S rather than USB. This mode is selected via modification of the configuration data stored in the flash device.

3. PRINCIPLES OF CONFIGURATION, CONTROL AND USAGE

The XVF3510 is intended to be used to provide Far-Field voice to a host system or processor in speech recognition and communication applications, either closely integrated to the main processor or as a USB accessory. As such the XVF3510 provides boot mechanisms from either an external QSPI flash or by the host processor over SPI interface.

To facilitate control in both boot configurations and to allow the specification of the default behaviour, the XVF3510 implements two mechanisms for control and parameterisation. The first is the Control Interface which is a direct connection between the host and the XVF3510 and is operational at runtime. The second is the Data Partition which is held in flash and contains configuration data to parameterise the XVF3510 on boot up. Both mechanisms have access to the full set of parameters and can both be used in the application to control and specify the behaviour of the device.

A host tool (vfctrl) is also provided provides command-line access to the control interface, allowing user access to all the configuration parameters of the XVF3510.

The following sections describe the following aspects of usage, configuration and control:

- Firmware release package.
- xTIMEComposer tools.
- vfctrl host command line tool.
- Configuration via a control interface.
- Configuration via the Data Partition.
- XVF3510 Development kits usage.

3.1. FIRMWARE RELEASE PACKAGE

There are two release packages available for the XVF3510, one for the XVF3510-UA and one for the XVF3510-INT. Both are available to download via the links in Section 5.2 below.

Release packages and firmware builds are identified via a version number, which follows the standard semantic version specification. The version number format is X.Y.Z, eg 4.0.0, and these numbers have the following meaning.

Table 3-1 Firmware version number structure

DIGIT	NAME	MEANING
X	Major version number	Significant release of the firmware. The control interface may not be backwards compatible with earlier versions
Y	Minor version number	New features added, but the control interface is backwards compatible with earlier host applications
Z	Patch version number	Bug fixes for incorrect functionality only. No change to host interface

The release version is contained in the file name of firmware file distribution and can also be read via the control interface using the GET_VERSION command.

Each package consists of several directories and files containing released firmware binaries, data-partition tools, host binaries and host source code. A simplified directory structure is shown below.

```
├── bin
├── data-partition
│   ├── images
│   └── input
└── host
    ├── Linux
    │   └── bin
    ├── MAC
    │   └── bin
    ├── Pi
    │   ├── bin
    │   └── scripts
    ├── Win32
    │   └── bin
    └── src
        ├── dfu
        ├── dpgen
        └── vfctrl
```

Further information about each component of the release is as follows:

3.1.1. "BIN" DIRECTORY

This directory contains the released firmware for the XVF3510. There are two copies of the firmware; one intended for loading from an external flash device and one for loading from an external host over SPI (XVF3510 is the slave). Please refer to the SPI Slave boot section of the datasheet for connections to the external boot source.

3.1.2. "DATA-PARTITION" DIRECTORY

The data partition contains configuration data for the XVF3510 firmware, implemented as a set of commands that are run at boot time. The data partition is created using input command source files and a set of tools which are described in the Data Partition section of this document. The contents of the data-partition directory are as follows:

The root directory contains default data partition image source files (`int.json` or `ua.json`) as well as the generic flash device specification `16mbit_12.5mhz_sector_4kb.spispec`, data partition generation scripts and short instructions about how to generate data partition binary files.

- The `images` subdirectory contains pre-generated data partition binary files generated from the default data partition image source file. These files are suitable for direct programming into the external flash along with the firmware, should the default settings be suitable.
- The `input` subdirectory contains short command sequences which are referenced by the data partition image source file when the data partition binary file is generated.

In addition, an `output` directory is created during the running of the data partition generation script which contains the newly generated data partition binary file.

3.1.3. "HOST" DIRECTORY

This directory contains files and utilities relating to the host. The various host utilities that perform parameter control, DFU and data partition generation are provided pre-compiled for Linux (ARM and x86), Windows and MacOS platforms. These binaries can be found in the `Linux`, `Pi`, `Mac` and `Win32` directories along with an additional script in for the Pi release called `send_image_from_rpi.py` which provides an example of sending an SPI boot image from the host.

The root of the `host` directory also contains scripts for unpacking packed signals which can be captured using the controls described in the signal routing section of this document.

Instructions for building the host utilities from the source are also provided in the same directory. The source files for the host utilities are contained in the `src` sub-directory allowing building, modification or integration into other projects.

Within this directory there are three further sub-directories `dfu`, `dpgen` and `vfctrl` which contain the source files (and dependent libraries) for the DFU, data partition generator and parameter control utilities.

3.2. REQUIRED TOOLS

In order to update the firmware, modify and regenerate Data Partitions and rebuild the host utilities the following tools are required.

3.2.1. XTIMECOMPOSER

The XMOS xTIMEcomposer contains a comprehensive suite of tools for compilation, debug and programming of XMOS devices. It is available to download https://www.xmos.ai/software-tools

NOTE: At the time of writing v14.4.1 of the xTIMEcomposer tools is recommend for XVF3510 operation.

More recent versions may be available, but unless specified on the xmos.ai website they will not have been tested and verified for operation with XVF3510.

Further information about the full tool suite, including installation instructions for different platforms is available here in the xTIMEcomposer user guide, available from https://www.xmos.ai/file/tools-user-guide

The XVF3510 Voice Processor is provided in two pre-compiled builds (-UA and -INT) and as such only requires the usage of the xTIMEcomposer programming tools, specifically xFLASH. This operates as a command-line application, to create the boot image, and if using flash, program the boot image to the attached device.

An XTAG debugger must be connected to the XVF3510 for flash programming operations. Refer to the Development Kit User Guide for information on using XTAG connections to XVF3510 development kits.

The basic form of the xFLASH command for flash image creation and programming with a data partition is as follows (note multiple lines have been used for clarity, but command should be executed on single line).

```
xflash --no-compression --boot-partition-size 1048576
       --factory [Application executable (.xe)]
       --data [Data partition description (.bin)]
```

For boot image generation over SPI from a host processor the following command is used:

```
Oculus Reparo (or the xFLASH equivalent)
```

- **Application executable (.xe)** - The .xe file is a boot image provided with a VocalFusion release package in one of the supported configurations (-UA or -INT product variants).
- **Data partition description (.bin)**- The .bin file is a data partition description either supplied in the release package (-UA or -INT) or customised as described later in this guide.

NOTE: Running xTIMEcomposer on macOS Catalina triggers a security issue. The resolution is detailed on the website here : https://www.xmos.ai/file/running-xtimecomposer-on-macos-catalina/

3.2.2. PYTHON 3

Some operations, such as running the SPI boot example on the Raspberry Pi, require the use of Python 3 (v3.7 onward is recommended). Python can be downloaded from http://python.org/downloads.

3.2.3. HOST BUILD TOOLS

In order to build the host utilities, the use of a platform-specific compiler is required.

WINDOWS

The host utilities are built with the *x86 Native Tools Command Prompt for VS* which is installed as part of the *Build Tools for Visual Studio.* This can be downloaded from Microsoft website (at the time of writing latest versions available here: https://visualstudio.microsoft.com/downloads/#build-tools-for-visual-studio-2019). It is important to ensure that the optional *C++ CMake tools for Windows* are included when setting up the installation.

LINUX

Depending on the distribution and version of Linux used, the following packages may need to be installed:

```
sudo apt-get install -y build-essential
sudo apt-get install -y pkg-config
sudo apt-get install -y libusb-1.0-0-dev
```

MAC OS

The XCode Command Line tools are required to build in on macOS. The following command can be used to install the tools.

```
xcode-select --install
```

3.3. COMMAND-LINE INTERFACE (VFCTRL)

To allow command-line access to the control interface on the XVF3510 processor, the **vfctrl** (**V**ocal**F**usion **C**on**tr**o**l**) utility is provided as part of the release package. This utility

Two versions of this utility are provided for control of the device (a third is used internally by the Data Partition generation process):

Table 3-2 vfctrl versions and platforms

VERSION	FUNCTION	HOST PLATFORMS SUPPORTED
vfctrl_usb	Control of XVF3510-UA over a USB interface	Windows, MacOS, Linux, Raspbian
vfctrl_i2c	Control of XVF3510-INT over i2c interface	Raspberry Pi (Raspbian) only

Source code for the utility is also provided for compilation for other host devices if required.

The general syntax of the command line tool, when used for device control, is as follows:

```
vfctrl_usb <COMMAND_VERB> [ arg 1] [arg 2]....[arg N] [# Comment]
```

The <COMMAND_VERB> is required and is used to control the parameters of the device. Commands can be read and write commands and are distinguished by the prefix 'GET_' and 'SET_' for parameter read and write respectively.

The available commands are described in detail in specific sections later in this document, and a summary table of all the parameters is provided in Appendix A.

Following the <COMMAND_VERB> there are a number of optional arguments [arg 1]..[arg N] which depend on the specific parameter. These are detailed in the command tables later in the document.

If the <COMMAND_VERB> is are GET_ command, the output of the operation is printed to the terminal as in the example below:

```
vfctrl_usb GET_GPI
GET_GPI: 13
```

The number and type of arguments depend on the command and these are detailed in the command tables. Arguments are integer numbers separated by a space. For setting some parameters that require floating-point data, the numbers have to be first converted to a Q format and then transferred as integers.

The specification of the Q format for representing floating-point numbers is given in Appendix H.

A secondary form of vfctrl is also available which provides information for developers

```
vfctrl [options]
```

Where [options] can be:

```
-h, --help           : List all command options
-d, --dump-params    : Print list of parameter values
-n, --no-check-version : Do not check version of firmware image
```

3.4. CONFIGURATION VIA CONTROL INTERFACE

The XVF3510 Voice Processor contains parameters which can be read and written by the host processor at run time. For information writing parameters at boot time for initial configuration, please see the section on the Data Partition later in this document.

The XVF3510 firmware is provided as two pre-compiled builds, -UA and -INT, which provide a parameter control mechanism over USB endpoint 0 and I2C respectively.

Device functions have controllable parameters for the audio pipeline, GPIO, sample rate settings, audio muxing, timing and general device setup and adjustment. Commands support either read using the GET_ prefix or write using the SET_ prefix. Controllable parameters may either be readable and writeable, read-only or write-only. Various data types are supported including signed/unsigned integer of either 8b or 32b, fixed point signed/unsigned and floating-point.

In addition, the -UA build includes volume controls for input (processed mic from XVF3510) and output (far-end reference signal). These are USB Audio Class 1.0 compliant controls and are accessed via the host OS audio control panel instead of the XVF3510 control interface. The volumes are initialised to 100% (0dB attenuation) on device power up, which is the recommended setting.

Ensure that the XVF3510-UA USB Audio input and output volume controls on the host are set to 100% (no attenuation) to ensure proper operation of the device. Some host OS (eg. Windows) may store volume setting in between device connections.

For a comprehensive list of parameters, their data types and an understanding of their function within the device please consult the User Guide section relevant to the function of interest, or Appendix A which summarises all the commands. The control utility can also be used by supplying the `-h` argument to the command line. This dumps a list of commands to the console along with a brief description of the function of each command. The remainder of this section will cover the generic operation of the control interface.

3.4.1. CONTROL OPERATION

The control interface works by sending a message from the host to the control process within the XVF3510 device. The time required to execute commands can vary, but most will respond within 30ms. Since the commands are fully acknowledged, by design, the control utility blocks until completion. This interface is designed to allow real-time tuning and adjustment but may stall due to bus access or data retrieval.

The control interface consists of two parts a host side application and the device application. These are briefly summarised below.

3.4.2. HOST APPLICATION

The example host applications, found in the `/host` directory in the Release Package, are command-line utilities that accept text commands and, in the case of a read, provides a text response containing the read parameter(s). Full acknowledgement is included in the protocol and an error is returned in the case of the command not being executed properly or handled correctly by the device.

Example host source code and makefiles for are provided in the release package for x86 Linux, ARM Linux (Raspberry Pi), Windows and Mac platforms along with pre-compiled executables to allow fast evaluation and integration. For more information refer to the *Building the host utilities from source code* section.

3.4.3. DEVICE APPLICATION

The device is always ready to receive commands. The device includes command buffering and an asynchronous mechanism which means that Endpoint 0, NACKing for USB or clock stretching for I2C is not required. This simplifies the host requirements particularly in the cases where clock stretching is not supported by the host I2C peripheral.

3.5. CONFIGURATION VIA DATA PARTITION

VocalFusion device flash firmware configuration is comprised of a Boot image and a Data Partition.

- The **Boot image** in the form of an .xe archive is the executable code. It is provided as part of the XVF3510-UA or XVF3510-INT Release Package. This configures the underlying operation of the device.
- The **Data Partition** configures a running Boot image instance at startup with a set of commands which are customisable for the specific application. This contains any command that can be issued at run-time via USB or I^{2}C, plus some more that are boot-time only. Pre-configured Data Partitions are supplied in the release packages for default operation.

This combination of Boot image and Data Partition allow the functionality of the processor to be configured and defined without requiring any modification or recompilation of base firmware. The commands discussed in subsequent sections can be stored in the Data Partition, for execution at startup redefining the default operation of the device.

3.6. XVF3510 DEVELOPMENT KITS

There are two variants of development kit available: VocalFusion development kit and VocalFusion development kit for Amazon AVS. These two kits share the same hardware but differ in the firmware which is pre-loaded into the flash memory of each kit. This is shown below:

Table 3-3 Development kit variants and firmware pre-loaded

DEVELOPMENT KIT	FIRMWARE LOADED	NOTES
VocalFusion development kit for Amazon AVS	XVF3510-INT	https://www.xmos.ai/vocalfusion-voice-interfaces/#3510-dev-kits Firmware named "XVF3510 I2S Firmware binary.xe" in early v0.12.0 release
VocalFusion development kit	XVF3510-UA	https://www.xmos.ai/vocalfusion-voice-interfaces/#3510-dev-kits Firmware named "XVF3510 Adaptive USB Firmware binary.xe" in early v0.12.0 release

NOTE: Users are recommended to check the website for the latest firmware update, and to follow the instructions below to update the stored firmware before operation.

The VocalFusion development kit for Amazon AVS, associated setup documents and host code enable users to build a complete Amazon Alexa endpoint with the addition of a Raspberry PI (not supplied). The XVF3510-INT connects to the Raspberry PI using I2S for audio, and I2C for control.

The signal flow through the development kit is shown below.

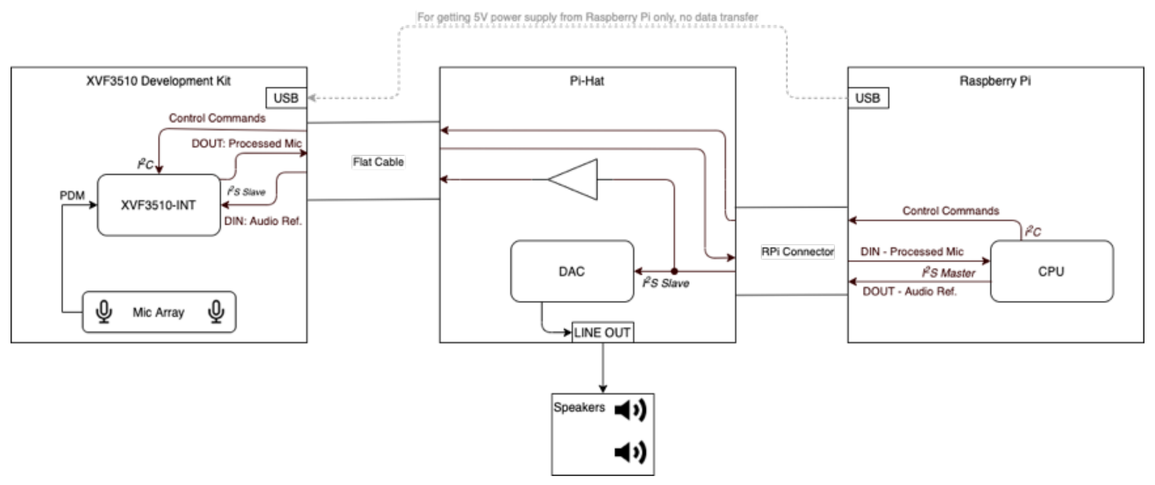


Figure 3-1 Signal flow of the VocalFusion development kit for Amazon AVS (XVF3510-INT)

The VocalFusion development kit provides a USB connection for audio and control. This can be used for evaluation of other ASRs or simply connected to a laptop or computer for audio analysis. In this configuration only the XVF3510 processor board and Microphone Array board are required. The signal flow is shown below:

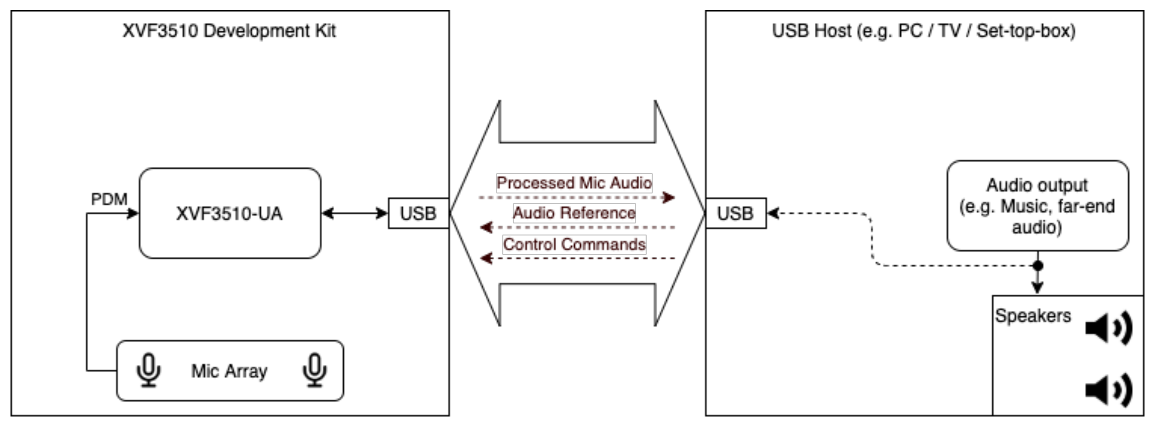


Figure 3-2 Signal flow of the VocalFusion dev kit (XVF3510-UA)

NOTE: In order to operate the latest firmware releases on development kits with revision numbers 1V0 and 1V1, a simple modification is required. This is detailed in a design advisory available on the website https://www.xmos.ai/file/xvf3510-development-kit-design-advisory

3.7. UPDATING THE FIRMWARE

As described above the hardware used on both development kits are identical allowing both firmware variants, -UA and -INT to operate correctly on either kit. The following steps should be used to update the firmware on the device.

1. Download the xTIMEcomposer tools from https://www.xmos.ai/software-tools and install onto the system that will be connected to the board to perform the update.
2. Download the latest version of the required XVF3510 Release Package from

XVF3510-UA Release	https://www.xmos.ai/file/xvf3510-ua-release
XVF3510-INT Release	https://www.xmos.ai/file/xvf3510-int-release

At the time of writing v4.2 is the latest released version.

3. Connect the XTAG Debugger to the system using the micro USB connection, plug the debugger into the XTAG connector, marked ‘DEBUG ONLY’, and power the board using micro USB connection marked USB on the XMOS processor board. (The power can be supplied via a USB connection from the system used to update, or a Raspberry Pi if used). The positions of these connectors are shown in the figure below:

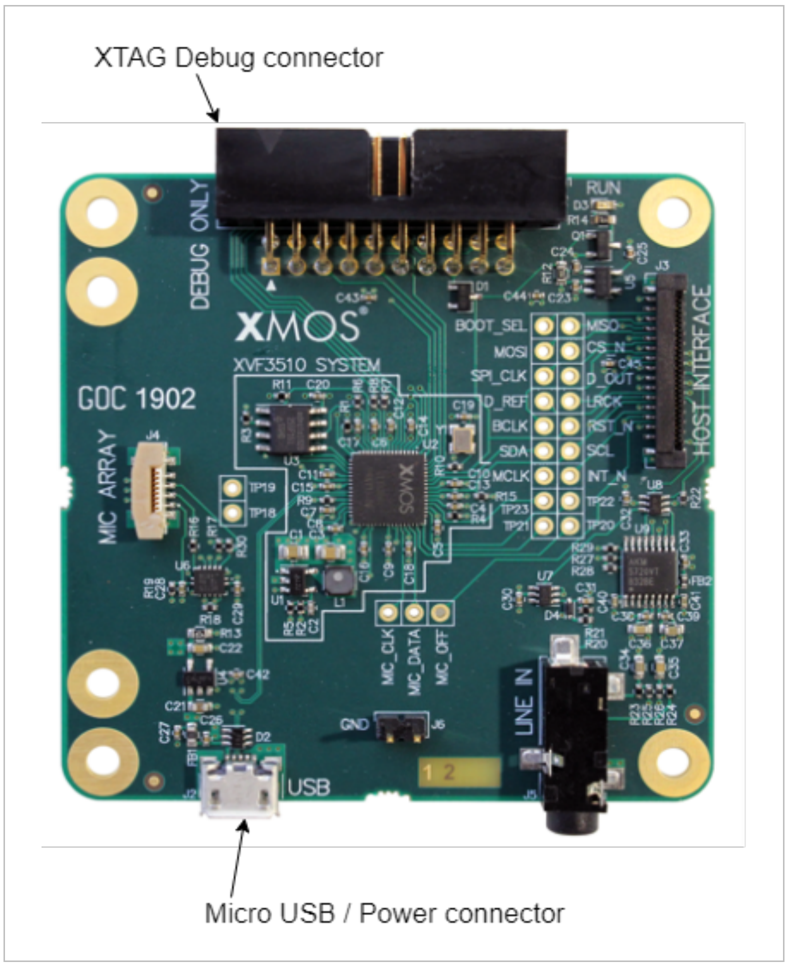


Figure 3-3 Location of Debug and power connectors on XVF3510 Development Kit Processor Board.

4. Open up an ‘xTIMEComposer Command Prompt’, or configure a terminal window using the appropriate setEnv script defined for the platform. For further information please consult the xTIMEcomposer User Guide (https://www.xmos.ai/file/tools-user-guide)

5. From the ‘xTIMEComposer Command Prompt’ or configured terminal window, navigate to the location of the Release Pack root directory. If installing the XVF3510-INT use the following command to re-flash the board with the updated XVF3510-INT firmware (note multiple lines have been used for clarity in command examples, but should be executed on single line):

```
xflash --no-compression --boot-partition-size 1048576
  --factory bin\app_xvf3510_int_vX_X_X.xe
  --data data-partition\images\data_partition_factory_int_vX_X_X.bin
```

If installing the XVF3510-UA use the following command to re-flash the board with the latest XVF3510-UA firmware:

```
xflash --no-compression --boot-partition-size 1048576
  --factory bin\app_xvf3510_ua_vX_X_X.xe
  --data data-partition\images\data_partition_factory_ua_vX_X_X.bin
```

Once the process has completed the following message indicates successful completion:

```
Site 0 has finished successfully.
```

The board is now configured with the latest version of XVF3510-INT or XVF3510-UA firmware.

3.8. OPERATION

The basic operation of both development kit options is described in the sections below.

3.8.1. XVF3510-INT AMAZON AVS DEMONSTRATION

The VocalFusion dev kit for Amazon AVS uses the XVF3510-INT to provide far-field voice to an AVS client running on a Raspberry Pi (not provided in development kit). The VocalFusion Development Kit for Amazon AVS Quick Start Guide and VocalFusion Development Kit for Amazon AVS User Guide detail the setup and usage instructions.

These guides can be downloaded from https://www.xmos.ai/file/xvf3510-dev-kit-setup-guides

The procedure for setting up the Amazon AVS SDK on the VocalFusion Development Kit can be found at

https://github.com/xmos/vocalfusion-avs-setup

Now the system will operate as an AVS endpoint using the XVF3510-INT as a Far-Field microphone, and Raspberry Pi to perform keyword detection and run the client. If the option to start the client automatically has been selected it will start on boot, otherwise, the following command should be from a Raspberry Pi terminal.

```
avsrun
```

Control and configuration of the XVF3510-INT is achieved using the I2C control interface. A VocalFusion Host Control application, (vfctrl_i2c), is provided pre-compiled and as source code for this purpose.

The following steps explain how to use the host control application.

1. Copy the host directory of the Firmware Release Pack to the Raspberry Pi.
2. Navigate from a terminal window to the copied host directory and execute the following command to list the supported commands and the general form of the utility usage.

```
./pi/bin/vfctrl_i2c --help
```

To verify that the system is setup correctly use the following command to list the I2C devices detected on the bus. The XVF3510-INT should appear at bus address 0x2C.

```
i2cdetect -y 1
```

If the XVF3510-INT is detected on the bus, but vfctrl_i2c returns the error:

```
rdwr ioctl error -1: No such device or address
```

check that the I2S clocks (MCLK, BCLK and LRCLK) are present and operational. Control requests can only be serviced when the I2S clocks are active.

3.8.2. XVF3510-UA USB CONNECTED DEMONSTRATION

The VocalFusion dev kit uses the XVF3510-UA to implement a USB Audio Class 1.0 (UAC 1.0) Far-field microphone, which can be connected to any USB host which can support UAC 1.0, such as laptop computers running Windows, Linux or macOS or Single Board Computer (SBC) systems running Linux or Android. The VocalFusion dev kit user guide and the VocalFusion development kit quick start guide detail the setup and usage instructions. These guides can be downloaded from https://www.xmos.ai/file/xvf3510-dev-kit-setup-guides

For completeness, the set-up procedure is also summarised below.

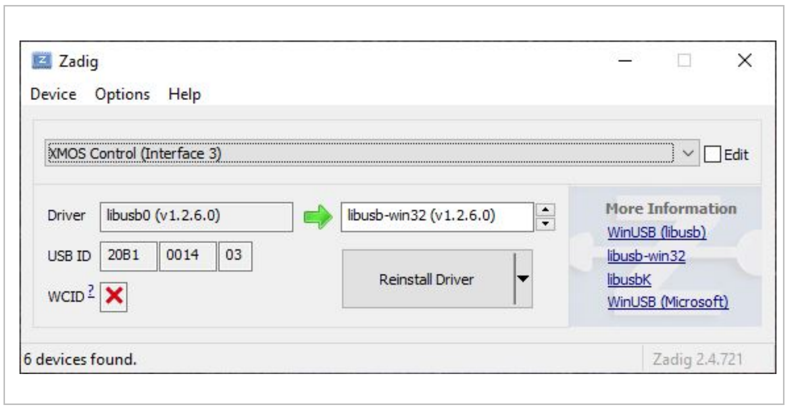
- Connect the USB Host (eg. laptop or SBC) to the XVF3510-UA via a USB cable, and connect speakers to the host processor system. Once connected the XVF3510 will enumerate as “XVF3510 (UAC1.0) Adaptive”.
- Next configure the output audio paths in the system such that both the speaker output and AEC reference paths (USB) are active. Details on how to enable this in Windows, MAC OS and Linux are provided in the User guide and quick start guides referenced above. Once configured, audio that is played out of the speakers will simultaneously be sent to the XVF3510 over USB providing a reference channel for the AEC.

- Now the audio capabilities of the system can be explored using an audio analysis package such as Audacity to record and playback audio to evaluate the far-field performance, noise suppression, and echo cancellation.

Control and configuration of the XVF3510-UA are achieved using via the control interface implemented over USB. A VocalFusion Host Control application, vfctrl_usb, is provided pre-compiled and as source code for this purpose.

For cross-platform support vfctrl_usb uses libusb. While this is natively supported in macOS and most Linux distributions, it requires the installation of a driver for use on a Windows host. Driver installation should be done using a third-party installation tool like Zadig (https://zadig.akeo.ie/). The following steps show how to install the libusb driver using Zadig:

- Connect the XVF3510 board to the host PC using a USB cable.
- Open Zadig and select XMOS Control (Interface 3) from the list of devices. If the device is not present, ensure Options -> List All Devices is checked.
- Select libusb-win32 from the list of drivers.



- Click Reinstall Driver.

Once installed the vfctrl_usb utility is ready to use. The following steps explain how to use the host control utility.

- Copy the host directory of the Firmware Release Pack to the host platform.

2. Navigate, from a terminal window, to the copied host directory and execute one of the following commands, depending on the specific platform, to list the supported commands and the general form of the utility usage.

For Linux hosts use:

```
./Linux/bin/vfctrl_usb --help
```

For macOS hosts:

```
./MAC/bin/vfctrl_usb --help
```

For Windows hosts:

```
.\WIN32\bin\vfctrl_usb --help
```

3.8.3. HOST UTILITIES

There are seven host utilities provided in the VocalFusion XVF3510 Release Package as pre-compiled utilities and also as source code to allow rebuilding other system architectures. The utilities are summarised below:

`data_partition_generator`, `vfctrl_json` - Uses .json configuration definition and generates binary Data Partitions for download to flash memory. `vfctrl_json` is used internally by the `data_partition_generator` but is referenced here for completeness.

`dfu_suffix_generator` - Adds DFU suffix to binary Boot Images and binary Data Partitions to protect the device from accidental DFU of incompatible image partition pair.

`dfu_usb`, `dfu_i2c` - DFU utilities for XVF3510-UA, XVF3510-INT respectively

`vfctrl_usb`, `vfctrl_i2c` - Vocal Fusion Control Utilities for the XVF3510-UA and XVF3510-INT respectively.

The pre-compiled versions are found in the following platform sub-directories within the host directory:

`/host/Linux` for Linux based systems

`/host/MAC` for macOS

`/host/Pi` for Raspbian based Raspberry Pi systems

`\host\Win32` for Windows platforms.

3.8.4. BUILDING THE HOST UTILITIES FROM SOURCE CODE

The source code for these utilities is provided in the following directory:

`\host\src`

The steps to build each utility are described in the Release Package here:

`\host\how_to_build_host_apps.rst`

3.8.5. DEFAULT OPERATION

The following table details the default configuration for the XVF3510-UA and XVF3510-INT firmware v4.2 after update using the procedure described above.

PARAMETER	DEFAULT - UA	DEFAULT -INT	CONFIGURABLE?
Version (x=patch version)	v4.2.x	v4.2.x	N
Reference input FROM host	USB UAC 1.0 48k samples/s PCM 16-bit resolution	I2S slave 48k samples/s PCM 32-bit resolution	Y (prior to microphone and I2S start up
Reference format	1 or 2 channel (Mono / Stereo)	1 or 2 channel (Mono / Stereo)	N
Processed audio output TO host	USB UAC 1.0 48k samples/s PCM 16-bit resolution	I2S bus 48k samples/s PCM 32-bit resolution	Y (prior to microphone and I2S start up
Audio format to host	2 channel - two different streams CH[0] - ASR CH[1] - Comms	2 channel - two different streams CH[0] - ASR CH[1] - Comms	Y
USB Product String	XVF3510 (UAC1.0) Adaptive	-n/a-	Y
USB Vendor ID	0x20B1 (8369)	-n/a-	Y
USB Product ID	0x0014 (20)	-n/a-	Y
USB Vendor String	XMOS	-n/a-	Y
USB Serial Number	null	-n/a-	Y
I2C address	N/A	0X2C	N
MCLK	24.576MHz OUTPUT	24.576MHz INPUT	Y
Acoustic Echo Canceller	Enabled	Enabled	Y
Automatic Delay Estimator	Activated once on startup	Activated once on startup	Y
Interference Canceller	Enabled	Enabled	Y
Noise suppressor	Enabled	Enabled	Y

4. XVF3510 FEATURES AND CONFIGURATION

This section describes in detail the features and configuration of the XVF3510 voice processor. It is organised into four sections which cover the main aspects of usage and configuration:

- Booting;
- Configuration and the Data Partition;
- Interfaces, Audio Routing and filtering;
- Far-field voice processing.

4.1. BOOTING

As demonstrated with the VocalFusion development kits the standard mechanism for booting is from an attached QSPI Flash device. This provides standalone operation, and persistent storage for configuration data. VocalFusion XVF3510 supports device firmware upgrade (DFU) over USB (-UA product variant) and I^{2}C (-INT product variant). Pre-compiled host utilities, and source code for reference, are supplied for performing DFU operations. The pre-compiled utilities can be found in the release package in one of the host architecture directories eg. `host\Win32\bin\dfu_usb.exe`, and the source code in `host\src\dfu`.

NOTE: While the functionality of the DFU is similar to the USB DFU specification, it has diverged to accommodate both USB and I2C operation and therefore is not compatible with compliant USB DFU tools.

The following sections discuss the structure of data within the flash memory, and operation of DFU.

4.1.1. FLASH STORAGE STRUCTURE

The structure of data within the VocalFusion XVF3510 is arranged to contain a factory image, a single upgrade image, device serial numbers and data partitions for both the factory and upgrade image. This is shown below.

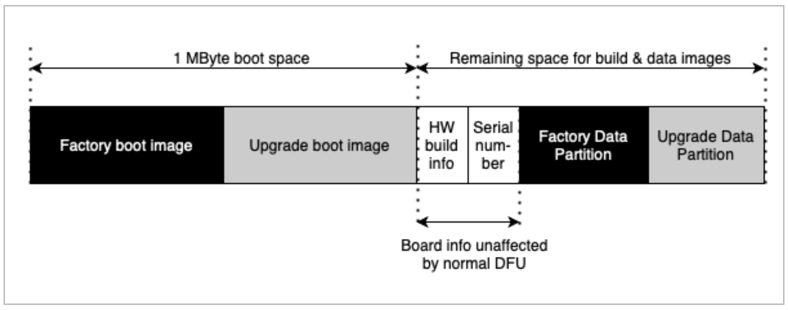


Figure 4-1 Flash data structure for VocalFusion XVF3510

- The **factory boot image** is the executable code for VocalFusion and is supplied in the Release Package in the `bin` directory. The file format is xe, which refers to XMOS Executable. This is written to the device via the XTAG debugger or through a bulk flash programming operation.
- The **upgrade boot image**, if present, is the executable code written to the flash memory via a DFU operation. Generation of the upgrade boot image is covered below.
- The **HW build info** is specified in the .json Data Partition file for the factory image and is written at the same time as the factory image and Data Partition. It is a unique identifier which is unaffected by subsequent DFU upgrade operations.
- The **Serial Number** is a custom field which can be programmed via USB and I2S control interfaces and remains untouched by the subsequent DFU operations.
- The **Factory and Upgrade Data Partitions** are the associated Data Partitions for the Factory and Upgrade images (where upgrade is present). They are written to flash in the same operation as the boot images. For more information on the generation and usage of Data Partitions see *Configuration and the Data Partition* section.

NOTE: Storage of only a single upgrade boot image and Data Partition pair are supported. Therefore, any Upgrade image applied will overwrite any existing upgrade image present.

A summary of the factory programming and field update process for flash-based systems is shown in the *Appendix I: Flash programming and update flow.*

4.1.2. PROGRAMMING THE FACTORY BOOT IMAGE AND DATA PARTITION

The process to program the Factory Boot image and Data Partition is described in the *Updating the firmware* section.

4.1.3. UPGRADE IMAGES AND DATA PARTITIONS

In order to be able to apply an Upgrade image to the device it must be programmed with a Factory Image and Data Partition as described above.

The DFU process requires the use of two utilities, `dfu_usb` or `dfu_i2c`, depending on the firmware variant, and `dfu_suffix_generator`. Precompiled versions are provided as part of the Release Package in the appropriate platform directory in `\host` (eg. `\host\Win32\bin`), and the source code for the DFU utility is provided in the `\host\src\dfu` directory. For more information on building the host applications refer to the build instruction file in `\host\how_to_build_host_apps.rst` in the Release Package.

In addition to the DFU utilities, the Upgrade image and Data Partition are required. These are provided in the Release Package in the `\bin` and `\data-partition\images`. Generation of custom Data Partitions is detailed in the *Configuration and the Data Partition* section. There are a number of stages required to prepare and execute a DFU to ensure are safe and successful update. These are detailed below.

GENERATION OF BINARY UPGRADE IMAGE

First, the Upgrade Image (.xe) needs to be converted to a binary format. Use `xflash` and the following command to convert the .xe image into a binary form (note multiple lines have been used for clarity in command examples, but should be executed on single line):

```
xflash --no-compression --noinq --factory-version 14.3
       --upgrade [UPGRADE_VERSION] [UPGRADE_EXECUTABLE] -o [OUTPUT_BINARY_NAME]
```

Even though the latest and recommended version of the tools is version 14.4.x, for legacy reasons we specify `--factory-version` value of 14.3. (The 14.3 value refers to boot loader API while 14.4 is toolchain version that retained the 14.3 API.)

NOTE: Should a different version of the tools be used, for example a future release, the version number should be noted such that an update image of compatible format can be created.

The upgrade version number is specified with `--upgrade`. The format is as follows 16bit 0xJJMP where J is major, M is minor and P is point.

For example, to create an upgrade Binary image for a -UA system, from the v4.0.0 Release Package use the following command:

```
xflash --no-compression --noinq --factory-version 14.3
       --upgrade 0x0400 app_xvf3510_ua_v4.0.0.xe -o app_xvf3510_ua_v4.0.0.bin
```

ADDITION OF DFU SUFFIX TO BINARY FILES

To prevent accidental upgrade of an incompatible image both the binary Upgrade image and the Data Partition binary must be signed using the provided `dfu_suffix_generator` which can be found pre-compiled in the host platform directory of the release package eg. `\host\MAC\bin`.

This mechanism embeds a structure into the binary files which can be read by the DFU tool to check that the binary data is appropriate for the connected device, prior to executing.

The general form of usage for the `dfu_suffix_generator` is as follows:

```
dfu_suffix_generator VENDOR_ID PRODUCT_ID [BCD_DEVICE] BINARY_INPUT_FILE
BINARY_OUTPUT_FILE
```

VENDOR_ID, PRODUCT_ID and BCD_DEVICE are non-zero 16bit values decimal or hexadecimal format, 0xFFFF bypassing verification of this field.

When building Upgrade images for XVF3510-UA devices, the USB Vendor Identifier (VID) and USB Product Identifier (PID) are added to the header and then checked by the DFU utility that the connected device matches. An error is reported by the tool if there is no match with the connected device.

For XVF3510-INT devices both Vendor and Product ID fields should be set to 0xFFFF for the generation. This instructs the DFU to bypass the checking as there is no equivalent to the USB identifiers for I2C systems. However, even though the checking is bypassed for the XVF3510-INT the suffix must be added to both Upgrade and Data partition files as the DFU utility checks the integrity of the binaries based on this information.

The following examples show how to add DFU Suffix to Update binaries for both XVF3510-INT and XVF3510-UA products.

For XVF3510-UA (default XMOS Vendor and XVF3510-UA product identifiers are used for illustration):

```
dfu_suffix_generator.exe 0x20B1 0x0014 app_xvf3510_ua_v4.0.0.bin boot.dfu
dfu_suffix_generator.exe 0x20B1 0x0014 data_partition_upgrade_ua_v4_0_0.bin
data.dfu
```

For XVF3510-INT:

```
dfu_suffix_generator.exe 0xFFFF 0xFFFF app_xvf3510_int_v4.0.0.bin boot.dfu
dfu_suffix_generator.exe 0xFFFF 0xFFFF data_partition_upgrade_int_v4_0_0.bin
data.dfu
```

NOTE: Extreme care must be taken if modifying the default Vendor and Product IDs through a Data Partition. If configuration from Data Partition fails the USB VID and PID will remain at their default values (VID=0x20B1, PID=0x0014) and DFU requests for signed files with modified will not be allowed.

PERFORMING DFU

The pre-compiled DFU utility is provided in the Release Package in the host architecture directory eg. `\host\Linux\bin`. For MAC, Linux and Windows the DFU_USB is provided, and for PI DFU_I2C is provided. The source code can be used to rebuild either version on the required platform.

The general form of dfu_usb utility is as follows:

```
dfu_usb [OPTIONS] write_upgrade BOOT_IMAGE_BINARY DATA_PARTITION_BIN

OPTIONS:    --quiet
            --vendor-id 0x20B1 (default)
            --product-id 0x0014 (default)
            --bcd-device 0xFFFF (default)
            --block-size 128 (default)
```

and the general form of the dfu_i2c utility is shown below:

```
dfu_i2c [OPTIONS] write_upgrade BOOT_IMAGE_BINARY DATA_PARTITION_BIN

OPTIONS:    --quiet
            --i2c-address 0x2c (default)
            --block-size 128 (default)
```

The two binary files passed to the utility, the boot image and data partition, must have the DFU suffix present otherwise the DFU utility will generate an error. Example DFU utility usage is shown for both XVF3510-UA and XVF3510-INT below.

For XVF3510-UA:

```
dfu_usb --vendor-id 0x20B1 --product-id 0x0014 write_upgrade boot.dfu data.dfu
```

and for XVF3510-INT:

```
dfu_i2c write_upgrade boot.dfu data.dfu
```

Once complete the following message will be returned and the device will reboot. In the case of XVF3510-UA the device will re-enumerate.

```
write upgrade successful
```

For verification that DFU has succeeded as planned, the vfctrl utility can be used to query the firmware version before and after update. For example, to query the version of XVF3510-UA the following command is used:

```
vfctrl_usb GET_VERSION
```

NOTE: The vfctrl utilities check the version number of the connected device to ensure correct operation. To suppress an error caused by a disparity in the version of vfctrl and upgraded firmware the --no-check-version option can be used with the utility.

4.1.4. FACTORY RESTORE

To restore the device to its factory configuration, effectively discarding any upgrades made, the same process as outlined above is followed but using a blank Boot Image and Data Partition.

This is the only way a restore can be initiated as the device does not have the ability to restore itself.

The same blank file can be used for both Boot Image and Data partition and can be generated using `dd` on MAC and Linux, and `fsutil` in windows as shown below:

An blank image can be created with a file of zeroes the size of one flash sector. In the normal case of 4KB sectors on a UNIX-compatible platform, this can be created as follows:

```
dd bs=4096 count=1 < /dev/zero 2>/dev/null blank.dfu
```

and for Windows systems:

```
fsutil file createNew blank.dfu 4096
```

The process outlined in the *Generation and application of Upgrade Image and Data Partition* section can now be followed using the blank.dfu file for both Boot Image and Data Partition.

4.1.5. BOOT IMAGE AND DATA PARTITION COMPATIBILITY CHECKS

The format of Data Partitions and Boot Images may change between version increments. Therefore to prevent incompatible Boot and Data Partitions from running and causing undefined behaviour, a field called `compatibility version` is embedded into the Data Partition. A running Boot Image checks its own version, against the `compatibility version` in the Data Partition before reading the partition data.

The version of the firmware should also be specified in the `--upgrade` argument of `xflash` when generating the Upgrade Image as described previously.

If the compatibility check fails, the booted image, which could be a factory image or an upgrade image will not read the Data Partition and will operate with its default settings (described in Default Operation section above). The Boot status is reported in the RUN_STATUS register which can be accessed via the vfctrl utility, for example:

```
vfctrl_usb.exe GET_RUN_STATUS
```

Successful Boot status is reported by either `FACTORY_DATA_SUCCESS` or `UPGRADE_DATA_SUCCESS` depending on which Boot Image was executed.

If unsuccessful the device will revert to a fail-safe mode of operation. The `RUN_STATUS` register can be queried for further debug information. The full list of RUN_STATUS codes are described in the *Appendix B: Boot Status codes (RUN_STATUS).*

NOTE: Fail safe mode uses default vendor ID of 0x20B1 (XMOS) and product ID of 0x14. In this event, host needs to be equipped with the ability to locate USB device under different IDs.

4.1.6. CUSTOM FLASH MEMORY DEVICES

The majority of QSPI flash devices conform to the same set of parameters which define the access and usage of flash devices. However, to support instances when the flash interface parameters are different, the following section explains how to define a custom flash interface.

Details of the flash device used to store the Boot Image and Data Partition data must be specified in two locations to ensure successful Factory programming and the ability to execute DFU to Upgrade the firmware. The Development kit uses a standard QSPI flash device which is representative of most 2MByte QSPI devices.

CUSTOM FLASH DEFINITION FOR FACTORY PROGRAMMING

During the Factory programming procedure, using the XMOS XTAG debugger, the specification of the flash device is used to create the loader which is responsible for downloading the Boot Image from flash and to the device. The flash specification is provided to `XFLASH`, as described in the *Updating the Firmware* section, using a SPISPEC file. A representative SPISPEC file, which supports the majority of QSPI flash devices and the Development Kits is provided in the Release Package here:

```
\data-partition\16mbit_12.5mhz_sector_4kb.spispec
```

This is a text file and must be modified with any differing parameters. An example .spispec file is shown in *Appendix C: Example .SPISPEC File Format* section.

CUSTOM FLASH DEFINITION FOR DATA PARTITION GENERATION

The SPISPEC file must also be included in the Data Partition, along with the Sector size so that DFU operations can be executed correctly.

NOTE: Due to the nature of the DFU function, it is critically important to test the execution of the DFU process in a target system prior to production manufacturing.

4.1.7. SPI SLAVE BOOT

This process was changed from V4.1 of the firmware

Both -UA and -INT configurations of XVF3510 have an SPI slave boot mode, in addition to the boot from flash mode. The SPI slave boot downloads the boot image in binary form, provided in the Release Package. This is illustrated using a Raspberry Pi and the Python script to manage the transfer as discussed below.

SPI BOOT OF XVF3510-INT USING DEVELOPMENT KIT AND RASPBERRY PI

Using the Development Kit, assembled as described in section *XVF3510-INT Amazon AVS demonstration*, and the XVF3510-INT Release Package available on the Raspberry Pi, a SPI boot can be executed by following the steps below:

1.Using a terminal console on the Raspberry Pi, navigate to the location of the XVF3510-INT Release Package.

2. Use the following command to execute the SPI boot process booting the XVF3510-INT firmware in the Release Package (replacing vX_X_X with the appropriate version number):

```
python3 host/Pi/scripts/send_image_from_rpi.py
     bin/app_xvf3510_int_spi_boot_vX_X_X.bin --delay
```

The device should be ready within 3 seconds.

3. Update the main clock in to PDM clock specific using the VocalFusion Control Utility vfctrl_i2c:

```
./host/Pi/bin/vfctrl_i2c SET_MCLK_IN_TO_PDM_CLK_DIVIDER 1
```

4. Configure any system specific settings using the VocalFusion Control Utility vfctrl_i2c.

5. Start the XVF3510 processing and interfaces by issuing the following commands over the VocalFusion control utility:

```
./host/Pi/bin/vfctrl_i2c SET_MIC_START_STATUS 1
./host/Pi/bin/vfctrl_i2c SET_I2S_START_STATUS 1
```

NOTE: Following an SPI boot the XVF3510 will not read any Data Partition that may be present in flash memory. This is the reason why step 3 is necessary, the command `SET_MCLK_IN_TO_PDM_CLK_DIVIDER` is included in the Data Partition for XVF3510-INT.

SPI BOOT OF XVF3510-UA USING DEVELOPMENT KIT AND RASPBERRY PI

NOTE: To illustrate the SPI Boot of XVF3510-UA on the development kit custom connection must be made between Pi Hat and XVF3510 Processor board. The connection is discussed in more detail in *Appendix D: SPI Boot custom connection.*

Using the XVF3510-UA Release Package available on the Raspberry Pi, a SPI boot can be executed by following the steps below:

1.Using a terminal console on the Raspberry Pi, navigate to the location of the XVF3510-UA Release Package.

2. Use the following command to execute the SPI boot process booting the XVF3510-UA firmware in the Release Package (replacing vX_X_X with the appropriate version number):

```
python3 host/Pi/scripts/send_image_from_rpi.py
    bin/app_xvf3510_ua_spi_boot_vX_X_X.bin --delay
```

The device should be boot within 3 seconds.

NOTE: The delay start mode is not available for XVF3510-UA.

4.2. CONFIGURATION AND THE DATA PARTITION

As described in a previous section, when using flash to boot the XVF3510 processor, the Data partition can be used to store commands which are executed immediately after boot-up to configure and define the functionality of the device. The following sections describe the definition of the Data Partition, how to generate, and the customisation for specific applications.

4.2.1. DATA PARTITION DEFINITION

PARTITION FILE STRUCTURE

The contents of a Data Partition are defined in a .json file which is passed to a generation script which forms the binary files used when flashing the device. The generation process is described below, after the definition .json file is described.

For the purpose of explanation consider the following example for a custom XVF3510-UA Data Partition:

```
{
    "comment": "",
    "spispec_path": "16mbit_12.5mhz_sector_4kb.spispec",
    "regular_sector_size": "4096",
    "hardware_build": "0xFFFFFFFF",
    "item_files": [
        {   "path": "input/usb_to_device_rate_48k.txt", "comment": "" },
        {   "path": "input/device_to_usb_rate_48k.txt", "comment": "" },
        {   "path": "input/usb_mclk_divider.txt", "comment": "" },
        {   "path": "input/xmos_usb_params.txt", "comment": "" },
        {   "path": "input/i2s_rate_16k.txt", "comment": "" },
        {   "path": "input/led_after_boot.txt", "comment":"" }
    ]
}
```

Comment pairs are provided for the .json configuration, but also the individual item files:

```
    { "comment": "" }
```

A running VocalFusion device needs to know size and geometry of its external QSPI flash in order to write firmware upgrades to it. This is added to a Data Partition in the form of a flash specification or SPI specification (See Appendix C for custom flash support)

```
    { "spispec_path": "16mbit_12.5mhz_sector_4kb.spispec" }
```

The Data Partition generation process aligns various sections onto flash sectors, and needs to know the sector size (this can be found in the flash device datasheet):

```
    { "regular_sector_size": "4096" }
```

Hardware build is a custom-defined, 32bit identifier written to flash along with the application firmware. It can be used to define a unique identifier for the hardware revision or other information which cannot be overwritten by subsequent updates:

```
    { "hardware_build": "0xFFFFFFFF" }
```

Item files which contain the commands to execute (format of item files described below). An optional comment field is provided:

```
    { "path": "input/usb_to_device_rate_48k.txt", "comment": "" }
```

NOTE: Because the generator is a Python script, the paths uses forward slashes irrespective of platform.

ITEM FILES

The item files contain the commands used to configure the system. The commands are simply added to the file in the same format as the command line control utility. For clarity, multiple item files can be included in the .json definition, each specifying a sub-set of commands relating to a particular function or aspect. Example item files for common configurations are provided in the `data-partition/input` directory of the release package. For example, the `agc_bypass.txt` item file bypasses the AGC for both output channels contains the following commands:

```
SET_ADAPT_CH0_AGC 0
SET_ADAPT_CH1_AGC 0
SET_GAIN_CH0_AGC 1
SET_GAIN_CH1_AGC 1
```

4.2.2. GENERATING A DATA PARTITION FOR CUSTOM APPLICATIONS

It is recommended that in order to create a custom Data Partition, an existing set of .json and item files is used as a template and modified as required. The release package contains example .json and item files for this purpose.

NOTE: The following process requires the use of Python 3. Installation is covered in *Required Tools* section.

The required additional control commands should be stored in an appropriately named text file inside the `data-partition/input` subdirectory. For example, a file named `aec_bypass.txt` could be added containing the collected commands:

```
SET_BYPASS_AEC 1
```

NOTE: Only commands which are required to be set with non-default values need to be included in the item file list.

These text files are then included in the custom JSON description.

In the above example, the `aec_bypass.txt` is added to to a JSON description, `bypass_AEC.json` as shown below:

```
    ...
    "item_files": [

        ...

        {
            "path": "input/aec_bypass.txt",
            "comment": ""
        }
        ...
    ]
    ...
```

NOTE: The execution order of the commands and input files can affect the behaviour of the device. Commands to configure USB and I^{2}S should be added at the beginning of the data image.

Finally, to generate the custom data partition, the command below should be run from the `data-partition` directory:

```
python3 xvf3510_data_partition_generator.py <build_type>.json
```

The generator script produces two data image files; one for factory programming and one for device upgrade in a directory named `output`.

For the above example these files will be called:

`data_partition_factory_<build_type>.bin`

and

`data_partition_upgrade_<build_type>.bin`.

These two binary files can be used to factory program or upgrade as described in *Updating the firmware* and *Generation and application of Upgrade Image and Data Partition* sections respectively.

A .JSON file is also produced for debugging purposes.

4.2.3. SERIAL NUMBER

The XVF3510 allows a 24 ASCII character long serial number to be stored in the external flash memory. This can be accessed using the VocalFusion Control application using the following commands (XVF3510-INT shown for example). To write to the serial number register use:

```
vfctrl_i2c SET_SERIAL_NUMBER "DEADBEEF"
```

and to read use:

```
vfctrl_i2c GET_SERIAL_NUMBER
```

USB DEVICE ENUMERATION (XVF3510-UA ONLY)

The XVF3510-UA additionally allows the Serial Number to be copied into the `iSerialNumber` field of the USB descriptor. As the host reads the USB descriptor on enumeration the command to copy the serial number must be present in the Data Partition. To illustrate this process the following commands must be incorporated into a Data Partition in the specified order (example assumes SERIAL_NUMBER field is already populated).

To set the USB configuration to use the serial number in the descriptor add the following lines, in this order, to the Data Partition:

```
SET_USB_SERIAL_NUMBER 1
```

To set the USB configuration to start enumeration:

```
SET_USB_START_STATUS 1
```

4.3. INTERFACES, AUDIO ROUTING AND FILTERING

The Following section describes, the interfaces, audio routing and filtering features of the XVF3510. Each section can be referred to in isolation, and describes all aspects relating to that feature.

4.3.1. USB INTERFACE

The following section details aspects that relate to the USB interface configuration and usage. This section only pertains to the XVF3510-UA variant of the processor.

The USB interface provides the host three end points:

- Adaptive USB Audio Class 1.0 endpoint for the transfer of Far-field voice to the host and AEC reference audio from the host.
- Vendor Specific Control allowing the host to control and parameterise the processor.
- Human Interface Device (HID) interrupt endpoint to signal the detection of events which have occurred on the GPIOs.

The USB Audio interface supports class compliant volume controls on both the input (processed microphone from XVF3510) and output (AEC reference) interfaces. These controls are accessed via the host OS audio control panels. They are initialised to 100% (0dB attenuation) on boot and this is the recommended setting for normal device operation.

By default the device will enumerate with the VID and PID shown below, but these can be configured using the Data Partition.

Table 4-1 Default USB Identification

USB IDENTIFICATION	VALUE
Vendor Identification (VID)	0x20B1
Product Identification (PID)	0x0014

The following section describes the parameters available to configure the USB interface behaviour.

USB CONFIGURATION

Due to the nature of the USB enumeration process, USB setup must be done using a Data Partition so that the configuration is complete prior to enumeration. The following table summarises the USB interface parameters which can be configured.

Table 4-2 USB configuration parameters

COMMAND	TYPE	ARGUMENTS	DEFINITION
SET_USB_VENDOR_ID GET_USB_VENDOR_ID	uint32	1	Set USB Vendor ID. See notes A, B.
SET_USB_PRODUCT_ID GET_USB_PRODUCT_ID	uint32	1	Set USB Product ID. See notes A, B.
SET_USB_BCD_DEVICE GET_USB_BCD_DEVICE	uint32	1	Set USB Device Release Number (bcdDevice). See notes A, B.
SET_USB_VENDOR_STRING GET_USB_VENDOR_STRING	uint8	25	Set USB Vendor string. See notes A, B.
SET_USB_PRODUCT_STRING GET_USB_PRODUCT_STRING	uint8	25	Set USB Product string. See notes A, B.
SET_USB_SERIAL_NUMBER GET_USB_SERIAL_NUMBER	uint32	1	Write only register, setting the behaviour of iSerialNumber field in USB descriptor (See notes A, B.): 1 - Load from Flash Serial Number 0 - Default to 0.
SET_USB_TO_DEVICE_RATE GET_USB_TO_DEVICE_RATE	uint32	1	Set sampling frequency of USB reference from USB host. Default is 48000 samples/sec. See notes A, B.
SET_DEVICE_TO_USB_RATE GET_DEVICE_TO_USB_RATE	uint32	1	Set sampling frequency of audio output to USB host. Default device_to_usb_rate is 48000 samples/sec. See notes A, B.
SET_USB_TO_DEVICE_BIT_RES GET_USB_TO_DEVICE_BIT_RES	uint32	1	Set bit depth of USB reference from USB host. Default usb_to_device_bit_res is 16 bits. See notes A, B.
SET_DEVICE_TO_USB_BIT_RES GET_DEVICE_TO_USB_BIT_RES	uint32	1	Set bit depth of audio output to USB host. Default device_to_usb_bit_res is 16 bits. See notes A, B.
SET_USB_START_STATUS GET_USB_START_STATUS	uint8	1	Start USB. Set as 1 as the last USB item in Data Partition. See notes A.

A: Command supported for Data Partition use only

B: Command must occur before SET_USB_START_STATUS 1

USB HID INTERFACE

A Human Interface Device (HID) is an electronic device with an interface which a human can use for control. Examples include a Personal Computer with a keyboard and mouse or a consumer appliance with control knobs, push buttons or a voice interface.

The XVF3510-UA uses the HID interface to inform the host system of events which have occurred on the General Purpose Inputs (GPI). The following section describes the setup of the GPI HID triggers.

HID REPORT GENERATION

The XVF3510 is able to send HID reports when an interrupt (logic edge transition event) on a GPI pin has been received. When interrupts are enabled using `SET_GPI_INT_CONFIG`, the interrupt bit is automatically serviced by the HID report generator. If an interrupt has occurred, then the sticky bit is immediately cleared and an HID report is generated. The HID features are described below:

- HID report for the assertion of GPI pin (positive edge) and report for the de-assertion (negative edge)
- The HID report type is generated with one of the following standard USB HID keycodes:
  - GUI Application Control Search (0x221)
  - GUI Application Control Stop (0x226)
  - Keyboard F23 (0x72)
  - Keyboard F24 (0x73)
- When no event has occurred, depending on “set idle” configuration by the host, it will either reply with a de-assert report (default) or NAK (set to idle by the host)

NOTE: HID idle behaviour is platform-specific and rarely the high-level application code will have any control over the settings. Linux, for example, typically silences the devices by issuing an indefinite idle (NAK report if no change). Other platforms such as MacOS, on the other hand, leave the device verbose by not issuing an idle (report always sent).

The HID function requires that a GPI pin is configured to generate interrupts on both edges.

The HID Report Descriptor used in XVF3510-UA translates the GPI pin interrupt into a HID Report asserting one of the predefined usages.

The HID Report has the format:

BIT	7	6	5	4	3	2	1	0
	F24	F23	AC Stop	AC Search	Reserved			

The corresponding bit equals 1 when a positive edge interrupt has been detected and zero where a negative edge interrupt has occurred. In order to configure the GPI pin that triggers the HID report, the SET_GPI_INT_CONFIG command is used.

For example, the following command configures GPI pin 0 to generate interrupts on both edges, which enables the HID report logic:

```
vfctrl_usb SET_GPI_INT_CONFIG 0 0 3
```

The first argument is a reserved value and should be set to 0. The second argument makes the command target pin IP_0. The third argument selects both edges for the interrupt. To make the device respond to the falling edge only with the value 1 and rising edge only with the value 2.

4.3.2. I2C SLAVE CONTROL INTERFACE (XVF3510-INT ONLY)

The XVF3510-INT implements an I2C slave interface for Control and Setup of the device. The interface conforms to the following specifications.

SPECIFICATION	VALUE
Maximum I2C operation speed	100kbps
I2C Slave Address	0x2C

4.3.3. GENERAL PURPOSE INPUT AND OUTPUT AND PERIPHERAL BRIDGING

The XVF3510 supports I/O expansion and protocol bridging over USB and I2C for the XVF3510-UA and XVF3510-INT respectively. This allows peripheral devices such as audio hardware connected to XVF3510 to be configured and monitored by the host.

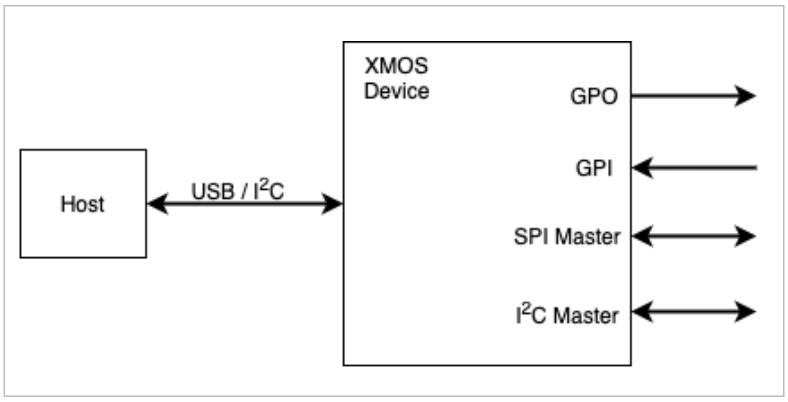


Figure 4-2 Device GPIO interfaces

- Four GPI channels (pins)
  - Direct read of port value
  - Rising, falling or Both edge capture with “sticky” bit which is cleared on read
  - Mode configurable per pin
- Four GPO channels (pins)
  - Direct write of entire port or pin
  - Active high or Active low
  - 500Hz PWM configurable between 0 and 100% duty
  - Blinking control supporting a sequence of 32, 100ms states
- SPI Master
  - 1Mbps SPI clock
  - Up to 128 Bytes SPI write
  - Up to 56 Bytes SPI read
- I2C Master (XVF3510-UA only)
  - 100kbps SCL clock speed
  - Register read/write (byte)
  - Up to 56 byte I2C read/write

The following sections describe the configuration and usage of each peripheral interface.

4.3.4. GPIO

There are four general input and four general output pins provided on the XVF3510.

Table 4-3 GPIO pin table

NAME	DESCRIPTION	I/O
IP_0	General purpose input	I
IP_1	General purpose input	I
IP_2	General purpose input	I
IP_3	General purpose input	I
OP_0	General purpose output	O
OP_1	General purpose output	O
OP_2	General purpose output	O
OP_3	General purpose output	O

GENERAL PURPOSE INPUTS

The following commands are available to read and control GPIs. Note that interrupt registers are set to 1 when an edge has been detected and 0 when no event has occurred. All interrupt registers are initialised to 0 on boot.

IP_0 is special in that, when interrupts are enabled, they are automatically serviced inside the chip and an HID report is generated over USB accordingly. See the *USB HID* section for further details.

The following parameters are available to interrogate and configure the GPI behaviour.

Table 4-4 General Purpose Input commands

COMMAND	TYPE	DIR	ARGS	DESCRIPTION
GET_GPI	uint32	READ	1	Read current level of all pins of the selected GPIO port. Pin 0 corresponds to the LSB of the port.
GET_GPI_INT_PENDING_PIN	uint32	READ	1	Read whether interrupt was triggered for selected pin. The interrupt pending register for the selected pin is cleared by this command.
GET_GPI_INT_PENDING_PORT	uint32	READ	1	Read whether interrupt was triggered for all pins on selected port. The interrupt pending register for the whole port is cleared by this command.
SET_GPI_PIN_ACTIVE_LEVEL	uint8	WRITE	3	Set the active level for a specific GPI pin. Arguments are <Port Index> <Pin Index> <0: active low, 1: active high>. By default, all GPI pins are set to active high.
SET_GPI_INT_CONFIG	uint8	WRITE	3	Sets the interrupt config for a specific pin. Arguments are <Port Index> <Pin Index> <Interrupt type 0=None, 1=Falling, 2=Rising, 3=Both>.
SET_GPI_READ_HEADER	uint8	WRITE	2	Sets the selected port and pin for the next GPIO read. Arguments are <Port Index> <Pin Index>.
GET_GPI_READ_HEADER	uint8	READ	2	Gets the currently selected port and pin set by a previous SET_GPI_READ_HEADER command.
SET_KWD_INTERRUPT_PIN	uint8	WRITE	1	Set gpi pin index to receive kwd interrupt on

COMMAND	TYPE	DIR	ARGS	DESCRIPTION
GET_KWD_INTERRUPT_PIN	uint8	READ	1	Read gpi pin index to receive kwd interrupt on

EXAMPLE: READING A GPIO PIN

For example, a read operation on XVF3510-UA is illustrated below. To read the level of pin 2 of the input port first set the port index (always 0 for XVF3510) and the pin index (2 in this case):

```
vfctrl_usb SET_GPI_READ_HEADER 0 2
```

Next perform the read:

```
vfctrl_usb GET_GPI
GET_GPI: 13
```

The returned value, 13 (b'1101), means pin index 1 (IP_1) is logic low and the other pins 0, 2 and 3 logic high.

EXAMPLE: CONFIGURING AND CAPTURING A FALLING AND RISING EDGE INTERRUPT

An example of configuration of a GPIs to capture edge events is discussed below (XVF3510-INT used for the example). First, configure IP_1 to trigger a falling edge interrupt and IP_2 to trigger a rising edge interrupt as shown:

```
vfctrl_i2c SET_GPI_INT_CONFIG 0 1 1
vfctrl_i2c SET_GPI_INT_CONFIG 0 2 2
```

For the example, IP_1 & IP_2 are connected to the same source, which is driving low. To check the ports for interrupts use the following commands:

```
vfctrl_i2c SET_GPI_READ_HEADER 0 0
vfctrl_i2c GET_GPI_INT_PENDING_PORT
> GET_GPI_INT_PENDING_PORT: 0
```

NOTE: The pin index specified in SET_GPI_READ_HEADER is ignored by GET_GPI_INT_PENDING_PORT

The result returned by the `GET_GPI_INT_PENDING_PORT` indicates that no transitions have occurred. Continuing the example, assume now that IP_1 & IP_2 are asserted high, and the port queried again:

```
vfctrl_i2c GET_GPI_INT_PENDING_PORT
> GET_GPI_INT_PENDING_PORT: 2
```

The result 2 (b'0010) shows that IP_2 has triggered on a rising edge. Rechecking the port status shows this event has been cleared.

```
vfctrl_i2c GET_GPI_INT_PENDING_PORT
> GET_GPI_INT_PENDING_PORT: 0
```

When IP_1 and IP_2 are driven low and the port status queried again:

```
vfctrl_i2c GET_GPI_INT_PENDING_PORT
> GET_GPI_INT_PENDING_PORT : 1
```

result (b'0001) shows that IP_1 has seen a falling edge interrupt.

GENERAL PURPOSE OUTPUTS

The following commands are available to write and control GPOs:

Table 4-5 General Purpose Output commands

COMMAND	TYPE	DIRECTION	ARGS	DESCRIPTION
SET_GPO_PORT	uint32	WRITE	2	Write a value to all pins of a GPIO port. Arguments are <Port Index> <Value>.
SET_GPO_PIN	uint8	WRITE	3	Write to a specific GPIO pin. Arguments are <Port Index> <Pin Index> <Value>.
SET_GPO_PIN_ACTIVE_LEVEL	uint8	WRITE	3	Set the active level for a specific GPO pin. Arguments are <Port Index> <Pin Index> <0: active low, 1: active high>. By default, all GPO pins are active high
SET_GPO_PWM_DUTY	uint8	WRITE	3	Set the PWM duty for a specific pin. Value given as an integer percentage. Arguments are <Port Index> <Pin Index> <Duty in percent>.
SET_GPO_FLASHING	uint32	WRITE	3	Set the serial flash mask for a specific pin. Each bit in the mask describes the GPO state for a 100ms interval. Arguments are <Port Index> <Pin Index> <Flash mask>.

NOTE: All GPOs have a weak pull-down (~30kΩ) during reset and initialised to logic low on device boot and will always drive the pin thereafter.

To illustrate usage of the GPOs the following section considers four common examples. Writing to a GPO pin, configuring a PWM output, generating a blink sequence and driving a three colour (RGB) LED.

The following commands toggle OP_2 high then low (XVF3510-UA shown for example):

```
vfctrl_usb SET_GPO_PIN 0 2 1
vfctrl_usb SET_GPO_PIN 0 2 0
```

To set all GPOs high and then low:

```
vfctrl_usb SET_GPO_PORT 0 15
vfctrl_usb SET_GPO_PORT 0 0
```

The PWM runs at a fixed 500Hz frequency designed to minimise visible flicker when dimming LEDs and supports 100 discrete duty settings to permit gradual off to fully-on control.

The following commands illustrate setting individual PWM frequencies on each output by setting GPO pins 0, 1, 2 and 3 to output 25%, 50%, 75% and 100% duty cycles respectively:

```
vfctrl_usb SET_GPO_PWM_DUTY 0 0 25
vfctrl_usb SET_GPO_PWM_DUTY 0 1 50
vfctrl_usb SET_GPO_PWM_DUTY 0 2 75
vfctrl_usb SET_GPO_PWM_DUTY 0 3 100
```

Setting a pin duty to 100% is the same as setting that pin to high.

Each GPO is driven from the LSB of an internal 32bit register, which is rotated by one bit every 100mS.

The figure below shows how the blinking sequence works:

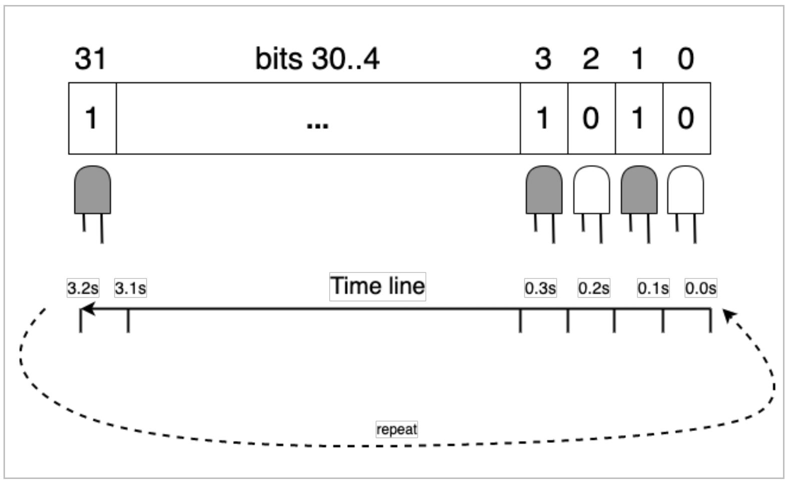


Figure 4-3 Use of 32 bit word is used to define the blinking function of GPO

The following commands configure the following:

- GPO pin 0 blinking, ON for 1.6 seconds, then OFF for 1.6 seconds, i.e. a period of 3.2 seconds;
- GPO pin 1 blinking, ON for 0.8 seconds, then OFF for 0.8 seconds, i.e. a period of 1.6 seconds;
- GPO pin 2 blinking, ON for 0.1 seconds, then OFF for 0.1 seconds, i.e. a period of 0.2 seconds;

```
vfctrl_usb SET_GPO_FLASHING 0 0 4294901760 # equivalent to pattern: xFFFF0000
vfctrl_usb SET_GPO_FLASHING 0 1 4042322160 # equivalent to pattern: xFF00FF00
vfctrl_usb SET_GPO_FLASHING 0 2 2863311530 # equivalent to pattern: xAAAAAAAA
```

Note that a GPO pin can be set to both a PWM duty cycle, and to flashing by issuing both a GPO_SET_PWM_DUTY instruction and a SET_GPO_FLASHING instruction for the same port and pin.

Where RGB LEDs are connected to three GPO pins (0 = Red, 1 = Green, 2 = Blue) automated colour sequencing can be programmed. For example, to colour cycle between Red-Yellow-Green-Cyan-Blue every 3.2 seconds:

```
vfctrl_usb SET_GPO_FLASHING 0 0 65535 # 0 x0000FFFF
vfctrl_usb SET_GPO_FLASHING 0 1 16776960 # 0 x00FFFF00
vfctrl_usb SET_GPO_FLASHING 0 2 4294901760 # 0 xFFFF0000
```

4.3.5. I2C MASTER PERIPHERAL INTERFACE (XVF3510-UA ONLY)

The XVF3510-UA variant provides an I^{2}C master interface which can be used as:

- a bridge from the USB interface, i.e. VFCTRL_USB commands can be used from the host to read and write devices connected to the I^{2}C Peripheral Port;
- a mechanism to initialise devices connected to the I^{2}C Peripheral Port by incorporating commands into the data partition (in the external flash), which are executed at boot time.

The interface supports:

- 100kbps fixed speed
- 7bit addressing only
- Byte I^{2}C register read/writes are supported.

The following table shows the commands for the configuration of the I^{2}C Master interface:

Table 4-6 I2C peripheral interface commands

COMMAND	TYPE	DIRECTION	NUM OF ARGS	NUMBER OF RETURNED VALUES	DEFINITION
SET_I2C_READ_HEADER	uint8	WRITE	3	0	Set the parameters to be used by the next GET_I2C, or GET_I2C_WITH_REG command. Arguments: 1: The 7-bit I2C slave device address. 2: The register address within the device. 3: The number of bytes to read.
GET_I2C_READ_HEADER	uint8	READ	0	3	Get the parameters to be used by the next GET_I2C, or GET_I2C_WITH_REG command. Returned values: 1: The 7-bit I2C slave device address. 2: The register address within the device. 3: The number of bytes to read.
GET_I2C	uint8	READ	0	56	Read from an I2C device defined by the SET_I2C_READ_HEADER command. Returned values: 1 to 56: The number of bytes read as defined by the SET_I2C_READ_HEADER command followed by additional undefined values. The number of bytes read from the I2C device when executing GET_I2C is set using SET_I2C_READ_HEADER
GET_I2C_WITH_REG	uint8	READ	0	56	Read from the register of an I2C device as defined by the SET_I2C_READ_HEADER command. Returned values: 1 to 56: The number of bytes read as defined by the SET_I2C_READ_HEADER command followed by additional undefined values. The number of bytes read from the I2C device when executing GET_I2C is set using SET_I2C_READ_HEADER
SET_I2C	uint8	WRITE	56	0	Write to an I2C slave device. Arguments: 1: The 7-bit I2C slave device address. 2: The number of data bytes to write (n). 3 to 56: Data bytes. All 54 values must be given but only n will be sent.

COMMAND	TYPE	DIRECTION	NUM OF ARGS	NUMBER OF RETURNED VALUES	DEFINITION
SET_I2C_WITH_REG	uint8	WRITE	56	0	Write to a specific register of an I2C slave device. Arguments: 1: The 7-bit I2C slave device address. 2: The register address within the device. 3: The number of data bytes to write (n). 4 to 56: Data bytes. All 53 values must be given but only n will be sent.

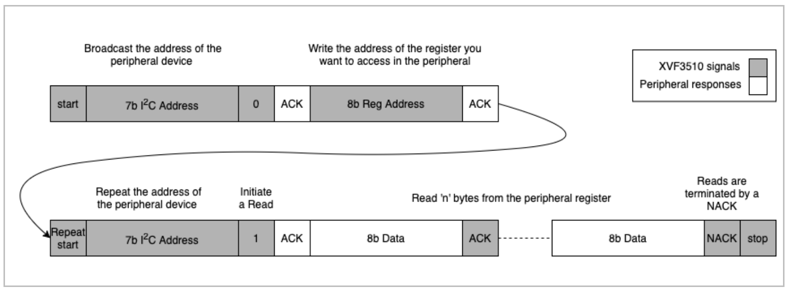

Broadcast the address of the peripheral device
Write the address of the register you want to access in the peripheral
XVF3510 signals
Peripheral responses
start
7b I²C Address
0
ACK
8b Reg Address
ACK
Repeat the address of the peripheral device
Initiate a Read
Read 'n' bytes from the peripheral register
Reads are terminated by a NACK
Repeat start
7b I²C Address
1
ACK
8b Data
ACK
8b Data
NACK
stop

Figure 4-4 I2C protocol for register reads

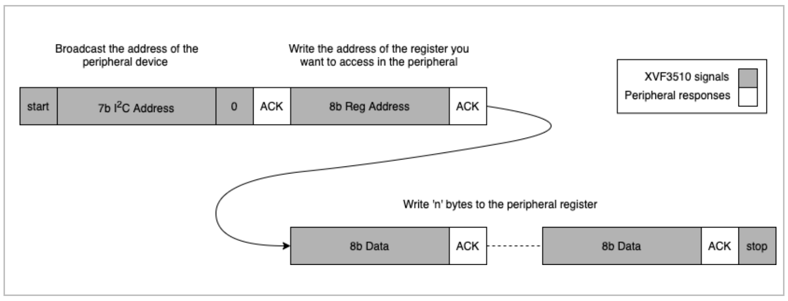

Broadcast the address of the peripheral device
Write the address of the register you want to access in the peripheral
XVF3510 signals
Peripheral responses
start
7b I²C Address
0
ACK
8b Reg Address
ACK
Write 'n' bytes to the peripheral register
8b Data
ACK
8b Data
ACK
stop

Figure 4-5 I2C protocol for register writes

- raw I²C read/writes may be performed.

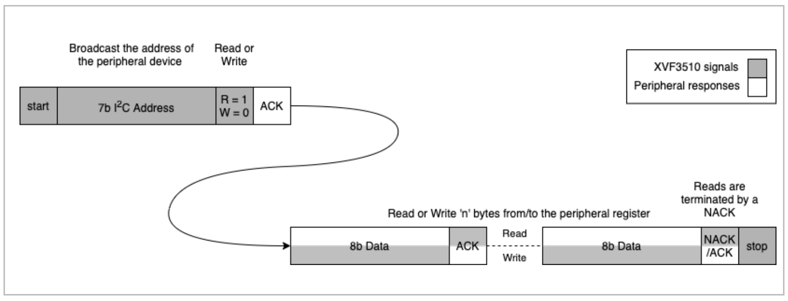

Broadcast the address of the peripheral device
Read or Write
XVF3510 signals
Peripheral responses
start
7b I²C Address
R = 1 W = 0
ACK
Read or Write 'n' bytes from/to the peripheral register
Reads are terminated by a NACK
8b Data
ACK
Read
Write
8b Data
NACK /ACK
stop

Figure 4-6 I2C protocol for raw reads and writes

USING I2C MASTER TO WRITE TO A DEVICE

Typically byte register read/writes are used to configure external I2C controlled hardware.

As an example, assume there is a device connected at address 0x40 (64) with three, single byte, registers. The following commands will write 77 to register 0, 48 to register 1 and 33 to register 2.

```
vfctrl_usb SET_I2C_WITH_REG 64 0 1 77 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
vfctrl_usb SET_I2C_WITH_REG 64 1 1 48 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
vfctrl_usb SET_I2C_WITH_REG 64 2 1 33 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

NOTE: The control protocol does not support variadic (variable number of) arguments. Hence, even when writing a single byte, the full number of arguments must bepassed. Unwritten values are ignored.

USING THE I2C MASTER TO READ FROM A DEVICE

To verify the previous I²C register write to register number 0 at address 0x40 (64), an I²C register read can be performed as follows:

```
vfctrl_usb SET_I2C_READ_HEADER 64 0 1
vfctrl_usb GET_I2C_WITH_REG
> 77 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

The byte read is the first of the 56 return values, which in this case, is 77. The following 55 values are undefined since the command only performed a read of 1 register.

4.3.6. SPI MASTER

The XVF3510_UA and XVF3510-INT variants provide an SPI master interface which can be used as:

- a bridge from the USB interface, i.e. VFCTRL_USB commands can be used from the host to read and write devices connected to the SPI Peripheral Port; and
- a mechanism to initialise devices connected to the SPI Peripheral Port by incorporating commands into the data partition (in the external flash), which are executed at boot time.

NOTE: From Version 4.1 the SPI Master peripheral interface is not available on XVF3510_UA and XVF3510_INT devices that have been SPI booted to prevent possible bus contention issues

The SPI master peripheral supports the following fixed specifications:

- Single chip select line
- 1Mbps fixed clock speed
- Supports either reads or writes. Duplex read/writes are not supported.
- Most significant bit transferred first
- Mode 0 transfer (CPOL = 0, CPHA = 0)

NOTE: The chip select is asserted a minimum of before 20ns the start of the transfer and de-asserted a minimum of 20ns after the transfer ends.

The SPI Master is controlled using the following commands.

Table 4-7 SPI peripheral interface commands

COMMAND	TYPE	DIR	ARGS	DESCRIPTION
GET_SPI	uint8	READ	56	Gets the contents of the SPI read buffer.
GET_SPI_READ_HEADER	uint8	READ	2	Get the address and count of next SPI read.
SET_SPI_PUSH	uint8	WRITE	56	Push SPI command data onto the execution queue.
SET_SPI_PUSH_AND_EXEC	uint8	WRITE	56	Push SPI command data and execute the command from the stack. Data will then be sent to SPI device.
SET_SPI_READ_HEADER	uint8	WRITE	2	Set address and count of next SPI read.

Reads of up to 56 Bytes at a time may be performed but writes of 128 Bytes at a time can be made by pushing multiple commands into a command stack and executing them in one go. The transaction is performed within a single chip select assertion.

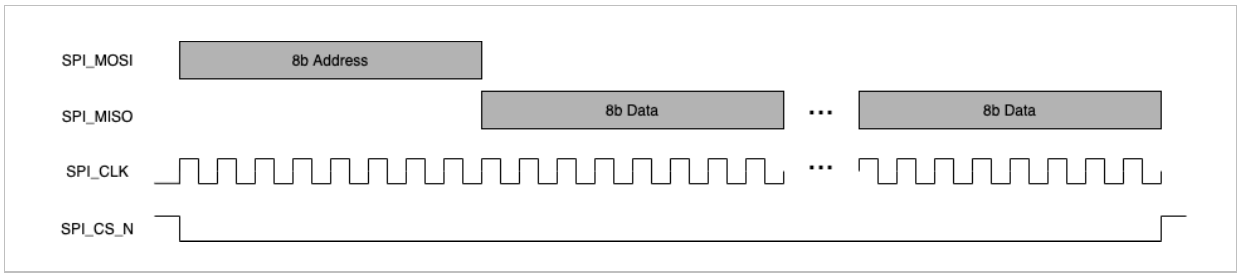


Figure 4-7 SPI peripheral, read sequence

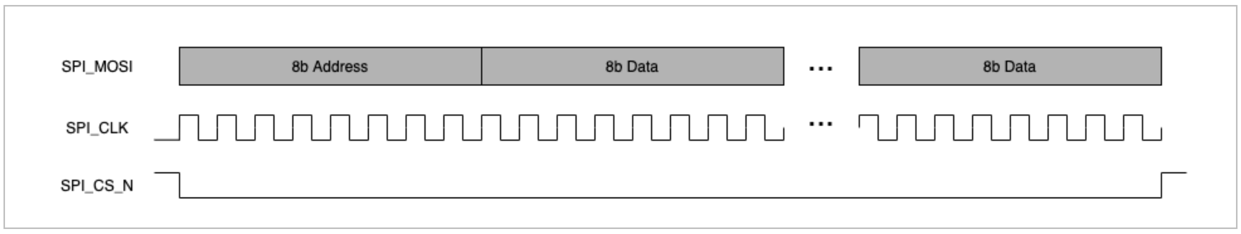


Figure 4-8 SPI peripheral, write sequence

The control protocol does not support variadic (variable number of) arguments. Hence, even when writing a single byte, the total number of arguments passed must be the maximum. Unwritten values are ignored.

See below examples.

The following example writes one byte of data (with value 122) to a control register as address 6.

```
vfctrl_i2c SET_SPI_PUSH_AND_EXEC 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 6 1 122
```

NOTE: All numbers are decimal. It is necessary to pad the payload to 56 bytes, which includes the address, length and data values. This is a requirement of the VFCTRL tool, the SPI interface itself will only transmit the valid data.

Transmitting more than 54 bytes of data is possible using the SET_SPI_PUSH command to queue up data, using multiple commands before the push is executed. The following example writes values 0 to 69 to address 100 (70 bytes in total) using command to push 56 data values into the queue, followed by a push the remaining 14 data words and then execute the transfer:

```
vfctrl_i2c SET_SPI_PUSH 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36
35
 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9
 8 7 6 5 4 3 2 1 0

vfctrl_i2c SET_SPI_PUSH_AND_EXEC  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 100 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56
```

To read one byte at address 6, which contains the value 122, we can do the following:

```
vfctrl SET_SPI_READ_HEADER 6 1
vfctrl GET_SPI
> GET_SPI: 122 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-> 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

To read 16 bytes from address 0, which all contain the value 33, we can do the following:

```
vfctrl SET_SPI_READ_HEADER 0 16
vfctrl GET_SPI
> GET_SPI: 33 33 33 33 33 33 33 33 33 33 33 33 33 33 33 0 0 0 0
-> 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-> 0
```

4.3.7. SIGNAL FLOW AND PROCESSING

Many of the parameters and functions of the XVF3510 may be controlled via the control interface. This control extends to being able to configure signal routing through the pipeline itself, providing flexibility useful in:

- Hardware testing of mics by monitoring the raw mic signal.
- Improving pipeline performance by filtering known noise sources at the raw mic input.
- Monitoring and debugging of reference signals and mic signals during development.
- Compensating for gain offset in reference signal.
- Supporting specific audio connectivity requirements such as obtaining the reference signal from I^{2}S.
- Inserting audio filtering where a speaker is connected downstream of the XVF3510 via I^{2}S.

The blocks supported are as follows:

- Signal Multiplexers. These allow dynamic selection (switching) of signals. The signals available depend on the multiplexer position.
- Gain Blocks. These are blocks that apply a variable bit shift (left or right) and, in the case of left shift, saturate in the case of overflow. Because they are shifters, the gain applied is a power of two.
- Filter Blocks. The filter blocks consist of two cascaded biquad units. Each of the five coefficients per stage is directly manipulated via the control utility.

The arrangement of the blocks, with respect to the device Input & Output and the XVF3510 audio processing pipeline, is shown in Figure 4-9 below:

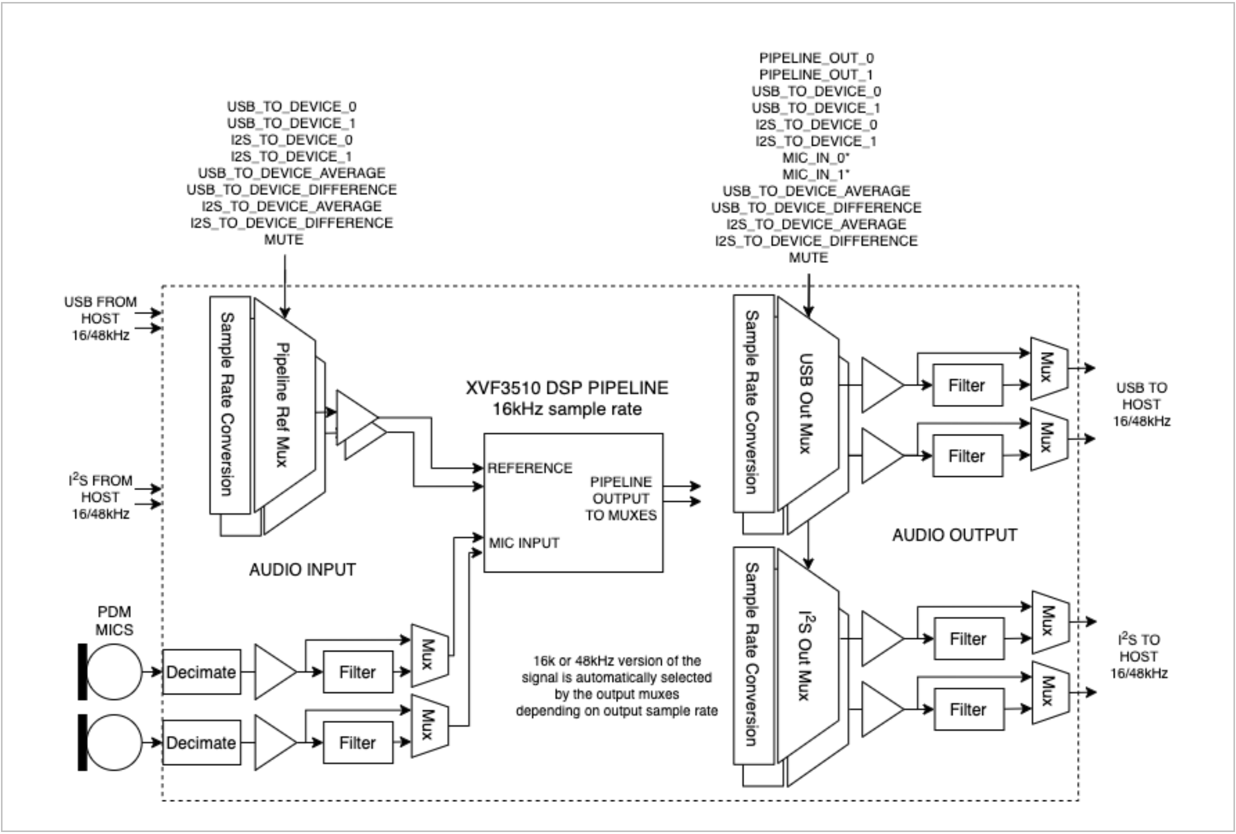


Figure 4-9 XVF3510 input, output and audio signal routing

The commands to control the audio multiplexes (Mux) blocks and detailed in Table 4-8 below and the source and destination index numbers are listed in Table 4-10 and Table 4-9 respectively

SIGNAL ROUTING AND SCALING

The following controls are provided for configuring the signal control blocks.

Table 4-8 I/O Mapping Commands

COMMAND	TYPE	ARGS	DEFINITION
SET_IO_MAP	uint8	2	Configures the two input switches and four output switches. See Destination and Source index table for valid argument options. arg1 <Destination Index> arg2 <Source Index>
SET_OUTPUT_SHIFT	int32	2	Sets the gain for each mux block. Select mux block Destination Index followed by shift (+ve is left, -ve is right shift) arg1 <Destination Index> arg2 <shift value>
GET_IO_MAP_AND_SHIFT	uint32	6 x 3	Get all 18 IO_MAP and OUTPUT_SHIFT values for all Destinations.
SET_MIC_SHIFT_SATURATE GET_MIC_SHIFT_SATURATE	uint32	2	Sets the gain on the raw mic signals before entering the pipeline. arg1 <shift value (left shift)> arg2 <saturate, enable if =1>

Where the Destination channels available to be mapped are referenced as follows:

Table 4-9   I/O Mapping Destination Indexes

CHANNEL (DESTINATION)	VALUE	DEFINITION
USB_FROM_DEVICE_0	0	USB channel 0 output from device to host
USB_FROM_DEVICE_1	1	USB channel 1 output from device to host
I2S_FROM_DEVICE_0	2	I2S channel 0 output from device
I2S_FROM_DEVICE_1	3	I2S channel 1 output from device
REF_TO_PIPELINE_0	4	reference channel 0 going into the pipeline
REF_TO_PIPELINE_1	5	reference channel 1 going into the pipeline

Sources available to be mapped to destination are referenced as follows:

Table 4-10   I/O Mapping Source Indexes

CHANNEL (SOURCE)	VALUE	DEFINITION
MUTE	0	Zeros are sent to the destination if this value is selected, which mutes the channel
USB_TO_DEVICE_AVERAGE	1	Average of USB input from host to device.
USB_TO_DEVICE_DIFFERENCE	2	Half of the difference between ch0 and ch1 of USB input from host to device.
I2S_TO_DEVICE_AVERAGE	3	Average of I2S input to device.
I2S_TO_DEVICE_DIFFERENCE	4	Half of the difference between ch0 and ch1 of I2S input to device.
PIPELINE_OUT_0	5	Pipeline output channel 0
PIPELINE_OUT_1	6	Pipeline output channel 1
USB_TO_DEVICE_0	7	USB input channel 0 from host to device
USB_TO_DEVICE_1	8	USB input channel 1 from host to device
I2S_TO_DEVICE_0	9	I2S input channel 0 to device
I2S_TO_DEVICE_1	10	I2S input channel 1 to device
MIC_IN_0	11	Ch0 Microphone input seen by the pipeline
MIC_IN_1	12	Ch1 Microphone input seen by the pipeline
PACKED_PIPELINE_OUTPUT	13	pack 16kHz pipeline output on 48kHz output (See Appendix J: Capturing packed samples for system integration for further information)
PACKED_MIC	14	pack 16kHz mic input to pipeline on 48kHz output (See Appendix J):
PACKED_REF	15	pack 16kHz reference input to pipeline on 48kHz output (See Appendix J)
PACKED_ALL	16	pack 1 channel of 16kHz mic, reference input and pipeline. When this option is used, the other channel of the same output also gets PACKED_ALL set in its IO map. (See Appendix J: Capturing packed samples for system integration for further information)

NOTE: The `MIC_IN_0` and `MIC_IN_1` signals are at 16kHz. If they are routed to a 48kHz output they will be sample repeated three times. No antialiasing filter is applied.

The following section illustrates how to use the IO mapping and scaling commands.

Using the `SET_IO_MAP` command, the user can choose the sources that get routed to the following 3 destinations -

- the USB output from device to host
- the I2S output from the device
- the reference going into the device

For instance, to route I2S channel 0 (= 9 as shown in the Source table) input to the device to USB channel 1 output from the device (= 1 as shown in the destination table), the command is:

```
vfctrl_usb SET_IO_MAP 1 9
```

where the first argument "1" refers to `USB_FROM_DEVICE_1` as shown in the destination table and the second argument "9" refers to `I2S_TO_DEVICE_0` in the source table.

Signal routing is also useful for hardware debugging of microphone or reference signal connection. As an example, the following command routes USB reference channel 0 from host to the USB audio output channel 0 of XVF3510:

```
vfctrl_usb SET_IO_MAP 0 7
```

A loopback of reference signal input XVF3510 and its audio output is formed. By playing back signal, e.g. a sine wave as the reference signal output from host, the user can verify if the signal is being received properly by XVF3510 through its audio output. If the audio signal recorded at host is different from the reference output, the user may check if the problem is caused by hardware connection failure or wrong data format.

Signal routing can also be used for debugging microphone signal:

```
vfctrl_usb SET_IO_MAP 1 12
```

The above command routes microphone channel 1 as the direct signal to XVF3510's USB audio output. Microphone signals can the be verified by recording XVF3510's audio output.

For XVF3510-UA, its I^{2}S master interface can be used for sending out different kind of signal shown in the source channel table while having USB outputs of processed audio. For example, the following command configures to send channels of mic, reference and pipeline outputs in 16kHz sampling frequency packed to 48kHz I^{2}S output:

```
vfctrl_usb SET_IO_MAP 2 16
vfctrl_usb SET_IO_MAP 3 16
```

By using Raspberry Pi with I^{2}S slave interface configured, the user can then capture synchronized signals of mic, reference and pipeline output. Observing these signals can be very useful for debugging. The packed signal can be unpacked to mic, reference and pipeline signal with 2 channels in each of them by using a Python script provided in the Release Package.

The `SET_OUTPUT_SHIFT` command can be used to specify a bit shift that is applied to all samples of a given target. For example, specifying:

```
vfctrl_usb SET_OUTPUT_SHIFT 2 4
```

applies a left shift of 4 bits on all samples output from the device on I2S channel 0 as 2^4=16x of gain. A negative shift value would imply a right bit shift for attenuation.

The `GET_IO_MAP_AND_SHIFT` command displays the IO mapping and the shift values for all targets.

Executing a `GET_IO_MAP_AND_SHIFT` command without having set any mapping or shifts explicitly shows the default mapping that is configured in firmware.

```
vfctrl_usb GET_IO_MAP_AND_SHIFT
GET_IO_MAP_AND_SHIFT:
target: USB_FROM_DEVICE_0, source: PIPELINE_OUT_0 output shift: NONE
target: USB_FROM_DEVICE_1, source: PIPELINE_OUT_1 output shift: NONE
target: I2S_FROM_DEVICE_0, source: PIPELINE_OUT_0 output shift: NONE
target: I2S_FROM_DEVICE_1, source: FAR_END_IN_0 output shift: NONE
target: REF_TO_PIPELINE_0, source: USB_TO_DEVICE_0 output shift: NONE
target: REF_TO_PIPELINE_1, source: USB_TO_DEVICE_1 output shift: NONE
```

GENERAL PURPOSE FILTER

The General Purpose filter blocks each comprise of two cascade biquad filters permitting configuration as bandpass, notch, low-pass, high-pass filters etc. By default, all filters are disabled (bypassed).

NOTE: A maximum of two output filters may be enabled simultaneously. Eg. Two channels of USB filtering or one I2S and one USB output. Exceeding this may cause audio glitching.

There is no restriction on input filters (mic and reference filters).

The filter coefficients are accepted in a floating-point format in a1, a2, b0, b1, b2 order directly from filter design tools such as https://arachnoid.com/BiQuadDesigner/index.html.

Support for the raw 32bit integer write/read is offered which directly accesses the internal representation. When using the raw control method, coefficients should be converted to Q28.4 format first and a_1 and a_2 need to be negated. See configuration parameters for more information.

The sample rate for filters on the input to the pipeline are always 16kHz whereas the output filters match the selected rate which may be either 16kHz or 48kHz, depending on system configuration. Ensure that the filter coefficients have been designed with the correct rate.

Note that, although potential numerical overflows are handled as a saturation, it is up to the designer to ensure no saturation occurs from the coefficients chosen to avoid non-linear behaviour of the filter. The implementation offers three bits of headroom (Q28.4) which is more than sufficient for most filters.

The coefficients are cleared to zero on boot.

The following table describes the commands for the configuration of the filters.

Table 4-11 Filter configuration parameters

COMMAND	TYPE	ARGUMENTS	DEFINITION
SET_FILTER_INDEX	uint8	1	Used as an index to point to which filter block that will be manipulated. output_filter_map_t below defines the filter block IDs.
GET_FILTER_INDEX	uint8	1	Retrieve the current filter index.
SET_FILTER_BYPASS	uint8	1	Bypass (1) means filter pointed to by the index is not enabled (default), 0 means enable the filter.
GET_FILTER_BYPASS	uint8	1	Retrieve the bypass status.
SET_FILTER_COEFF	float	10 (5x2)	Set 5 x 2 biquad coefficients in a floating-point format in the order a1, a2, b0, b1, b2. Coefficient a0 is assumed to be 1.0. If it is not, divide all coefficients by a0.
GET_FILTER_COEFF	float	10 (5x2)	Retrieve the floating-point representation of the coefficients in the order a1, a2, b0, b1, b2.
SET_FILTER_COEFF_RAW	int32	10 (5x2)	Set 5 x 2 biquad coefficients in Q28.4 format for the filter pointed to by the index. See note above in Filter Blocks section about the format.
GET_FILTER_COEFF_RAW	int32	10 (5x2)	Retrieve the Q28.4 representation of the coefficients. See note above in Filter Blocks section about the format.

Filter output indexes available to be used with filter setting commands (output_filter_map_t):

CHANNEL	VALUE	DEFINITION
FILTER_USB_FROM_DEVICE_0	0	USB channel 0 from device to host (Left)
FILTER_USB_FROM_DEVICE_1	1	USB channel 1 from device to host (Right)
FILTER_I2S_FROM_DEVICE_0	2	I2S channel 0 from device (Left)
FILTER_I2S_FROM_DEVICE_1	3	I2S channel 1 output from device (Right)
FILTER_MIC_TO_PIPELINE_0	4	16kHz mic channel 0 going into the pipeline
FILTER_MIC_TO_PIPELINE_1	5	16kHz mic channel 1 going into the pipeline
FILTER_REF_TO_PIPELINE_1	6	16kHz reference channel 0 going into the pipeline (Left)
FILTER_REF_TO_PIPELINE_1	7	16kHz reference channel 1 going into the pipeline (Right)

While setting the index or bypass control will always be safe, there is a small chance that the coefficients may be partially updated halfway through a filter operation. For this reason, the filter state is also cleared following updating to ensure that any possibility of instability is reduced. It is up to the user to ensure that the coefficients provided result in a stable filter configuration.

See Appendix G for a worked example on filter definition.

4.4. FAR-FIELD VOICE PROCESSING

4.4.1. PDM MICROPHONE INTERFACE

The PDM microphone interface converts Pulse Density Modulation (PDM) audio input from the microphones to Pulse Code Modulation (PCM) format allowing further processing. The PDM microphone interface consists of the physical pins connecting to the two microphones and a series of filters resulting in a 16kHz PCM, two-channel output stream suitable for far-field voice processing. Please refer to the datasheet for the physical and electrical details of the PDM pins.

The processing consists of four filter stages:

- Decimate by 8 FIR filter to 384kHz
- Decimate by 4 FIR filter to 96kHz
- Decimate by 6 FIR filter to 16kHz
- DC Blocking, single-pole IIR filter

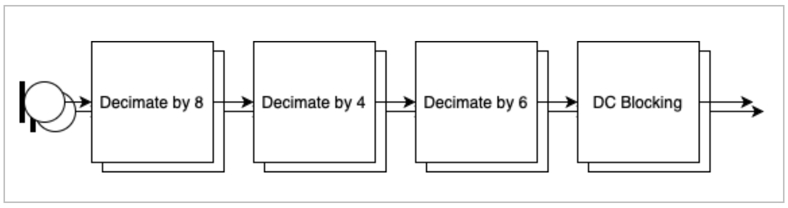


Figure 4-10 PDM microphone processing steps

The PDM microphone interface uses 32-bit internal processing to provide very low distortion with a specification exceeding -110dB THD+N with a 140dB dynamic range.

The frequency response of the FIR filter has a stopband attenuation of at least 70dB with a passband ripple of less than 0.9dB and a passband of 6.8kHz. The total group delay from pin to the XVF3510 audio pipeline input is 1.125 milliseconds.

A DC blocking filter is placed at the end of the PDM microphone interface pipeline and is tuned to have a 5Hz -6dB point and removes any DC offset present in the PDM input.

The output from the PDM microphone interface may optionally be shifted or attenuated providing a ‘power of two’ gain control. Saturation may be applied in the case that the gain is greater than one.

By default, the gain block shift is set to zero (a gain of $2^0 = 1$) and this is the recommended setting for normal use.

The PDM interface control parameters are shown below:

Table 4-12 Microphone commands

COMMAND	TYPE	VALUE	DESCRIPTION	NOTES
SET_MIC_SHIFT_SATURATE	uint32	arg1 <shift value (left shift)> arg2 <saturate, enable if !=0>	Write the gain (power of 2) on the raw mic signals before entering the audio pipeline.	
GET_MIC_SHIFT_SATURATE	uint32		Read the gain (power of 2) on the raw mic and Saturate Enable signals before entering the audio pipeline.	

4.4.2. AUTOMATIC ECHO CANCELLATION (AEC)

This process uses the stereo audio from the product as a reference signal to model the echo characteristics between each loudspeaker and microphone, caused by the acoustic environment of the device and room.

The AEC uses four models to continuously remove echoes in the microphone audio input created in the room by the loudspeakers. The models continually adapt to the acoustic environment to accommodate changes in the room created by events such as doors opening or closing and people moving about.

An illustration of echo paths in two sizes of room are shown below.

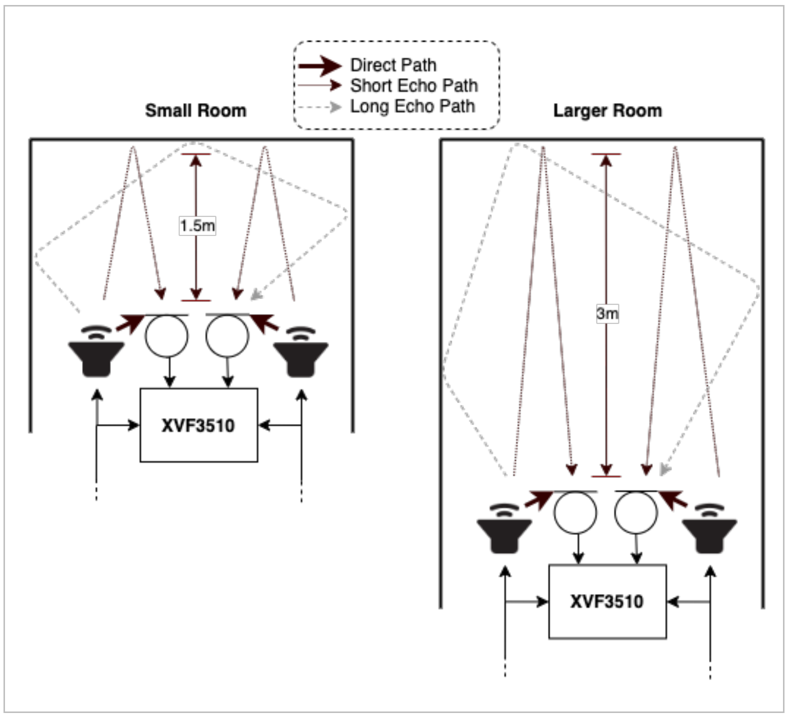


Figure 4-11 Echo paths from the speakers to the microphones

After reset, or when echo paths change due to a change in the environment, the AEC will re-converge. Echo Return Loss Enhancement (ERLE) can be used to indicate the degree of convergence on the AEC filters as shown below.

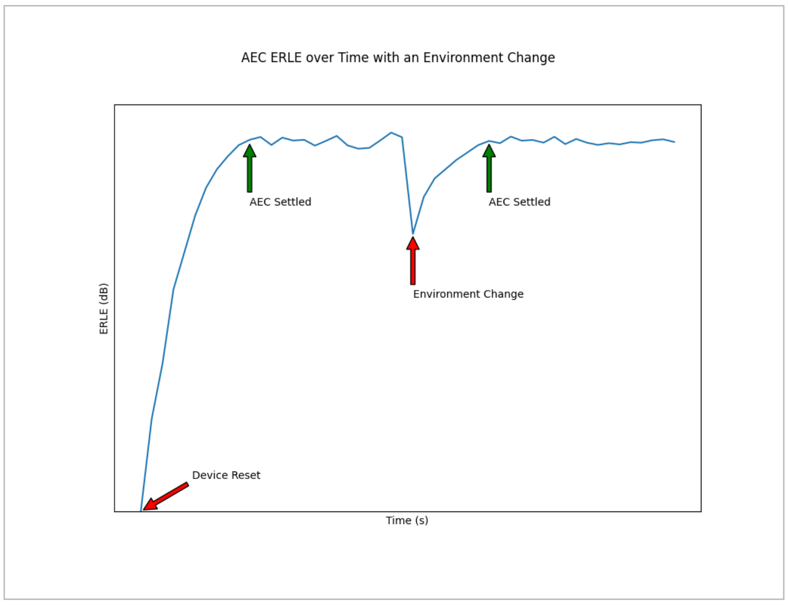


Figure 4-12 Settling time of the AEC shown using an ERLE plot

For optimal AEC settling-time performance, the volume of the speakers must be linearly proportional to the level of the reference audio sent to the XVF3510. If the volume of the speakers change without the level of the reference changing by the same linear factor, the AEC will respond as if the environment has changed such that all echo paths have increased/decreased energy, and will therefore incur a settling time in the AEC.

The Alternative Architecture (described in the *Alternative Architecture mode (ALT_ARCH)* section*)* selectively extends the AEC filters to accommodate highly reverberant environments.

The configuration parameters for the AEC are shown below:

Table 4-13 Useful Automatic Echo Canceller (AEC) commands

COMMAND	TYPE	VALUE	DESCRIPTION	NOTES
GET_BYPASS_AEC SET_BYPASS_AEC	uint32	[0,1] 0 = AEC bypass disabled (default) 1 = AEC bypass enabled	Get/Set AEC bypass parameter. If set to one, AEC processing is disabled.	A
SET_ADAPTATION_CONFIG_AEC GET_ADAPTATION_CONFIG_AEC	uint32	[0, 1, 2] 0 = Auto adapt (default) 1 = Force adaptation ON 2 = Force adaptation OFF	Sets AEC adaptation configuration. If AEC is set to bypass then setting the adaptation config has no effect.	B
GET_ERLE_CH0_AEC	float		Get AEC ERLE for channel 0	
GET_ERLE_CH1_AEC	float		Get AEC ERLE for channel 1	C
RESET_FILTER_AEC			This command resets all AEC filters.	

[A] When the Alternative Architecture (ALT_ARCH) mode is enabled (default), AEC bypass state will be overwritten and so should not be used. The GET command remains functional. For more information see the *Alternative Architecture (ALT_ARCH)* section.

[B] If Automatic Delay Estimation is enabled, these parameters will be overwritten and so should not be used. The GET commands remain functional. For more information see the *Automatic Delay Estimation & Correction (ADEC)* section.

[C] When the ALT_ARCH mode is enabled, there is only valid ERLE data available on CH0. In this mode the GET_ERLE_CH1_AEC will report NaN.

NOTE: The AEC operates on acoustic paths modelled in the AEC tail length. The Automatic Delay Estimation and Correction module handles delays between microphone and loudspeaker introduced by the equipment, for instance receiving the reference ahead of it actually being played out of the loudspeakers.

4.4.3. AUTOMATIC DELAY ESTIMATION & CORRECTION (ADEC)

The ADEC module automatically corrects for possible delay offsets between the reference and the loudspeakers.

Echo cancellation is an adaptive filtering process which compares the reference audio to that received from the microphones. It models the reverberant time of a room, i.e. the time it takes for acoustic reflections to decay to insignificance. This is shown in the figure below (the red “Acoustic echo path delay”).

The time window modelled by the AEC is finite, and to maximise its performance it is important to ensure that the reference audio is presented to the AEC time aligned to the audio being reproduced by the loudspeakers. The diagram below highlights how the reference audio path delay and the audio reproduction path may be significantly different, therefore requiring additional delay to be inserted into one of the two paths, correcting this delay difference.

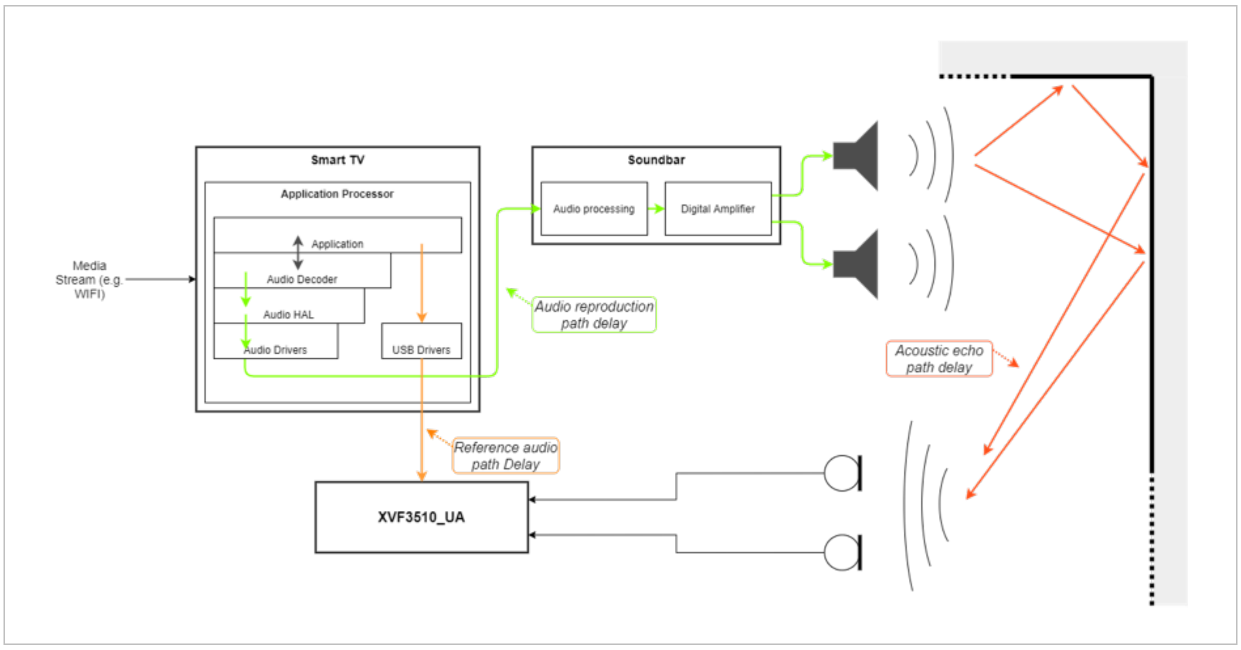


Figure 4-13 ADEC use case diagram

The functional blocks in the ADEC are shown below:

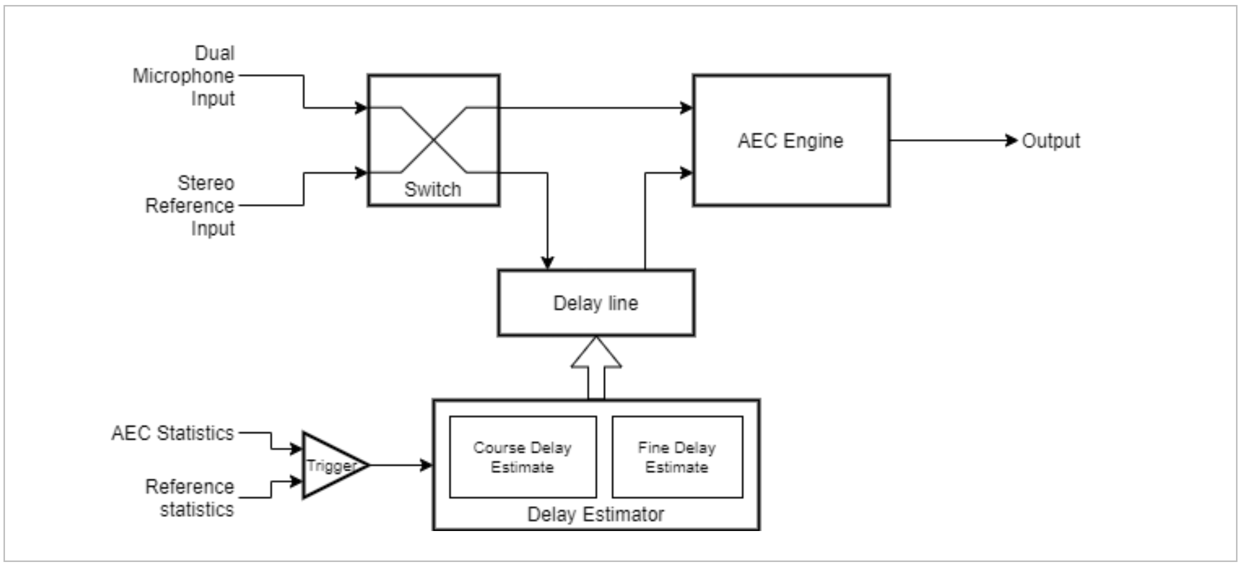


Figure 4-14 ADEC block diagram

Delay corrections may be applied to either microphone or reference path, to cope with reference or loudspeaker being ahead of the other, accomplished by switching the delay into either the microphone channel or the reference channel.

Automatic delay estimation is triggered at power-up, or if the host system configuration changes. The process will not begin until the reference signal is present and has sufficient energy.

The delay estimation process supports two stages:

- Fine delay estimation, during which AEC adaption is paused. Fine delay estimation searches over a limited delay window to detect small changes in delay. If the delay correction is not resolved the coarse estimation is triggered.
- Coarse delay estimation, re-purposes the AEC to detect larger delays. During estimation, the AEC does not perform cancellation. Once the delay is detected, and delay correction made the AEC restarts and converges based on the delayed signals.

Possible causes that may trigger coarse estimation:

- Device reset (if initial delay estimation is enabled, default: enabled)
- Host changing applications
- Large volume changes between the reference and the loudspeaker play-back
- User equipment changes, such as switching from TV audio output to playing the audio through a soundbar

Possible causes that may trigger fine estimation:

- Host toggling between audio devices (such as using a voice assistant application and listening to music at the same time) This is typically only seen in USB configurations
- Host processor performance leading to poor USB buffer management

The characteristics and specification of the ADEC function is shown below:

Table 4-14 ADEC characteristics

NAME	VALUE	DESCRIPTION
Maximum delay correction	± 150ms	The maximum delay that can be added to either the microphone channel or the reference channel.
Coarse estimation time	With good reference SNR: 2-5 seconds	During this time AEC is disabled. Note that estimation will not start unless reference is available and loudspeakers are playing back.

NAME	VALUE	DESCRIPTION
Fine estimation time	<1 second. If fine-grain estimation fails, coarse-grain estimation is triggered.	During this time adaptation in the AEC is disabled. Note that estimation will not start unless reference is available and loudspeakers are playing back.

The configuration commands are shown below:

Table 4-15 Automatic Delay Estimator parameters

COMMAND	TYPE	VALUE	DESCRIPTION	NOTES
GET_DELAY_SAMPLES SET_DELAY_SAMPLES	uint32	[0 .. 2399]	Change the number of samples of input delay at the sample rate 16kHz. The delay is applied to either the reference or the microphone in put according to the delay direction. This provides a maximum delay of +/- 150mS.	A
GET_DELAY_DIRECTION SET_DELAY_DIRECTION	uint32	[0,1] 0 - Delay the reference input (de fault) 1 - Delay the microphone input	Select the direction of input delay. i.e. it is applied to either the microphone input path or the reference signal path.	A
GET_DELAY_ESTIMATE	uint32	[0 .. 7200]	Get an estimate of the number of samples of delay on the reference input at a sample rate of 16kHz. This value is valid only when a coarse-grain delay estimation is in progress, and is offset by the maximum length of the delay buffer (2400 samples). Add 2400 samples to this value to get the absolute delay estimate.	
SET_ADEC_ENABLED GET_ADEC_ENABLED	uint32	[0, 1] 0 - ADEC disabled 1 - ADEC enabled	Enable automatic coarse-grain delay control. If automatic fine-grain delay control is enabled (SET_LOCKER_ENABLED 1), this parameter is overridden by a state machine internal to the firmware.	
GET_ADEC_MODE	uint32	[0,1] 0 - Normal AEC mode 1 - delay estimation in progress	Get the status of coarse-grain delay estimation.	
SET_MANUAL_ADEC_CYCLE_TRIGGER	uint32		Trigger a delay estimation cycle. The default behaviour in firmware is to trigger a coarse-grain delay estimation cycle when the far end reference is detected for the first time after device reset. This is done irrespective of whether automatic coarse-grain delay control is enabled or disabled. To disable this initial delay estimation, set SET_MANUAL_ADEC_CYCLE_TRIGGER =0 in the data partition. For all other times, if coarse-grain delay estimation is disabled, the SET_MANUAL_ADEC_CYCLE_TRIGGER can be used to force a coarse-grain delay estimation cycle.	

XMOS
Bringing technology to life

COMMAND	TYPE	VALUE	DESCRIPTION	NOTES
GET_AEC_PEAK_TO_AVERAGE_RATIO	float		Get current AEC filter coefficients peak to average ratio. If this value is above 4, the AEC has a "good" peak to average ratio.	
SET_LOCKER_ENABLED GET_LOCKER_ENABLED	uint32	[0,1] 0 - Automatic fine-grain delay control disabled 1 - Automatic fine-grain delay control enabled	Enable automatic fine-grain delay control. If enabled, the fine-grain delay control state machine overrides the setting for automatic coarse-grain delay control, so the SET_ADEC_ENABLED control command shouldn't be used.	
SET_LOCKER_DELAY_SETPOINT_ENABLED GET_LOCKER_DELAY_SETPOINT_ENABLED	uint32	[0,1] 0 - delay setpoint disabled (default) 1 - delay setpoint enabled	Set the delay setpoint enabled flag. When enabled, if the fine-grain delay estimator is unable to find the correct delay, then instead of triggering a coarse-grain delay estimate it sets the delay to a user defined value. This can reduce recovery time after a delay change. Before setting SET_LOCKER_DELAY_SETPOINT_ENABLED to 1, make sure that the delay value and direction are set using SET_LOCKER_DELAY_SETPOINT_SAMPLES and SET_LOCKER_DELAY_SETPOINT_DIRECTION commands	
SET_LOCKER_DELAY_SETPOINT_SAMPLES GET_LOCKER_DELAY_SETPOINT_SAMPLES	uint32	default: 0	Set the number of samples of delay that the automatic fine-grain delay control sets if SET_LOCKER_DELAY_SETPOINT_ENABLED is set to 1, and the fine-grain estimator fails to converge to a delay.	
SET_LOCKER_DELAY_SETPOINT_DIRECTION GET_LOCKER_DELAY_SETPOINT_DIRECTION	uint32	[0,1] 0 - delay the Reference input (default) 1 - delay the Mic input	Set the direction of input delay that the automatic fine-grain delay control sets if SET_LOCKER_DELAY_SETPOINT_ENABLED is set to 1, and the fine-grain estimator fails to converge to a delay.	
GET_LOCKER_STATE	str	"BOTH_WAIT" "LOCKER_SEARCH" "ADEC_TRIGGERED" "DELAY_PROPAGATING"	Get the current state of automatic fine-grain delay control state machine.	

[A] When either of automatic coarse-grain or fine-grain delay control systems are enabled, this value will be overwritten, therefore the SET commands should not be used. GET commands remain valid.

4.4.4. INTERFERENCE CANCELLER

The Interference Canceller (IC) suppresses static noise from point sources such as cooker hoods, washing machines, or radios for which there is no reference audio signal available. When an internal Voice Activity Detector (VAD) indicates the absence of voice, the IC adapts to remove noise from point sources in the environment. When the VAD detects voice, the IC suspends adaptation which maintains suppression of the interfering noise sources previously adapted to.

The IC only operates on the ASR channel from the pipeline output. The communications output channel optionally has a beamformer which fixes a region of interest directly in front, perpendicular to the plane of the two microphones.

The following table describes the configuration parameters for the Interference Canceller.

Table 4-16 Interference Canceller (IC) parameter

COMMAND	TYPE	VALUE	DESCRIPTION	NOTES
SET_BYPASS_IC GET_BYPASS_IC	uint32	[0,1] 0 = IC bypass disabled (default) 1 = IC bypass enabled	Set IC bypass parameter. If set to one, IC processing is bypassed.	A
SET_CH1_BEAMFORM_ENABLE GET_CH1_BEAMFORM_ENABLE	uint32	[0,1] 0 = Passthrough IC input channel 1 onto IC output channel 1 1 = Beamformed output on IC output channel 1 (default)	Enable beamformed output on IC output channel index 1.	
RESET_FILTER_IC			This command resets the IC filter.	

[A] If Alternative architecture mode (ALT_ARCH) is enabled (default), the IC bypass state will be dynamically changed by the firmware. Do not use the SET command. The GET command remains functional.

4.4.5. NOISE SUPPRESSOR (NS)

The Noise Suppressor (NS) suppresses noise from sources whose frequency characteristics do not change rapidly over time. This includes diffuse background noise and stationary noise sources.

The following table describes the settings for the Noise Suppressor.

Table 4-17 Noise Suppressor (NS) commands

COMMAND	TYPE	VALUE	DESCRIPTION
SET_BYPASS_SUP GET_BYPASS_SUP	uint32	[0,1]	Set suppressor bypass parameter. If set to one, the suppressor, which contains the noise suppression stages is bypassed. 0 - suppressor bypass disabled (default) 1 - suppressor bypass enabled
SET_ENABLED_NS GET_ENABLED_NS	uint32	[0,1]	Set noise suppression enabled parameter within the suppressor. If set to one, the noise suppression stage within suppressor is enabled. Changing this parameter only takes effect if the suppressor is not bypassed. 0 - noise suppression disabled 1 - noise suppression enabled (default)

4.4.6. AUTOMATIC GAIN CONTROL (AGC) AND LOSS CONTROL

The Automatic Gain Control (AGC) can dynamically adapt the audio gain, or apply a fixed gain such that voice content maintains a desired output level. The AGC uses an internal Voice Activity Detector to normalise voice content and avoid amplifying noise sources and applies a soft limiter to avoid clipping on the output.

The desired output level of voice content is defined by an upper and lower threshold. If a voice signal is outside of the upper and lower threshold then the gain will adapt accordingly. If the voice signal is within the upper and lower threshold then the gain will remain constant.

The rate at which the gain increases or decreases per audio frame can also be configured. The gain increment value must be greater than 1, whilst the gain decrement value must be below 1. When the gain is adapting, the current gain value is multiplied by either the increment or decrement value to calculate the gain value to be applied on the next audio frame.

The Loss Control process improves the subjective audio quality by attenuating any residual echo of the reference far-end audio. It is designed to be used on the communications channel. In cases where there is both far-end echo and near-end audio then the attenuation is reduced, allowing listeners to interrupt each other. The Loss Control relies on the Automatic Echo Canceller in order to classify and attenuate residual far-end echo.

The following table details the configuration parameters for the AGC.

Table 4-18 Automatic Gain Control (AGC) parameters

COMMAND	TYPE	VALUE	DESCRIPTION
SET_ADAPT_CH0_AGC SET_ADAPT_CH1_AGC GET_ADAPT_CH0_AGC GET_ADAPT_CH1_AGC	uint32	[0,1]	Set to enable gain adaptation in the AGC for channel 0 or 1. 0 - adaptation disabled for the channel 1 - adaptation enabled for the channel
SET_LC_ENABLED_CH0_AGC SET_LC_ENABLED_CH1_AGC GET_LC_ENABLED_CH0_AGC GET_LC_ENABLED_CH1_AGC	uint32	[0,1]	Set Loss Control to be enabled in the AGC for channel 0 or 1. 0 - Loss Control disabled for the channel 1 - Loss Control enabled for the channel
SET_GAIN_CH0_AGC SET_GAIN_CH1_AGC GET_GAIN_CH0_AGC GET_GAIN_CH1_AGC	Q16.16	[0..32767]	Set the linear gain parameter to be applied in the AGC for channel 0 or 1. Values are linear. Default: 500
SET_MAX_GAIN_CH0_AGC SET_MAX_GAIN_CH1_AGC GET_MAX_GAIN_CH0_AGC GET_MAX_GAIN_CH1_AGC	Q16.16	[0..32767]	Set the maximum gain threshold in the AGC for channel 0 or 1. Values are linear. Default: 1000
SET_UPPER_THRESHOLD_CH0_AGC SET_UPPER_THRESHOLD_CH1_AGC GET_UPPER_THRESHOLD_CH0_AGC GET_UPPER_THRESHOLD_CH1_AGC	Q1.31	[0..1]	Set the upper threshold for desired voice level. Values are in range 0 to 1 (full-scale) and must be greater than the lower threshold of the channel.
SET_LOWER_THRESHOLD_CH0_AGC SET_LOWER_THRESHOLD_CH1_AGC GET_LOWER_THRESHOLD_CH0_AGC GET_LOWER_THRESHOLD_CH1_AGC	Q1.31	[0..1]	Set the lower threshold for desired voice level. Values are in range 0 to 1 (full-scale) and must be lower than the upper threshold of the channel.

COMMAND	TYPE	VALUE	DESCRIPTION
SET_INCREMENT_GAIN_STEPSIZE_CH0_AGC SET_INCREMENT_GAIN_STEPSIZE_CH1_AGC GET_INCREMENT_GAIN_STEPSIZE_CH0_AGC GET_INCREMENT_GAIN_STEPSIZE_CH1_AGC	Q16.16	[1..32767]	Set the rate at which the gain increases. This value is applied on a per-frame basis when voice content is detected.
SET_DECCREMENT_GAIN_STEPSIZE_CH0_AGC SET_DECREMENT_GAIN_STEPSIZE_CH1_AGC GET_DECREMENT_GAIN_STEPSIZE_CH0_AGC GET_DECREMENT_GAIN_STEPSIZE_CH1_AGC	Q16.16	[0..1]	Set the rate at which the gain decreases. This value is applied on a per-frame basis when voice content is detected.

4.4.7. ALTERNATIVE ARCHITECTURE MODE (ALT_ARCH)

The Alternative Architecture mode, when enabled, improves Echo Cancellation performance in reverberate environments. It operates by re-configuring the audio pipeline by switching out either the AEC or the IC, depending on the energy in the AEC reference signal, to recover resources to be used to increase the specification of the remaining pipeline.

The two audio pipeline configurations are summarised below:

- **ALT_ARCH disabled** ALWAYS apply echo-cancelling AND interference cancelling; or
- **ALT_ARCH enabled** apply ONLY echo-cancelling when a reference signal is available, otherwise ONLY apply interference cancelling

The figure below expands the implementation details of the alternative mode switching. Multiplexers permit the AEC and/or the IC to be bypassed. When the IC is bypassed, only a single channel from the AEC is used, allowing it to be reconfigured, extending the filters to support a longer tail length. An internal module which collects statistics about the reference is used to dynamically control these multiplexers and memory allocation during runtime.

NOTE: Manually bypassing the IC using the Control Interface does not apply the memory reallocation.

The figure below highlights the audio signal path when the Alternative Architecture is disabled (ie. standard operation).

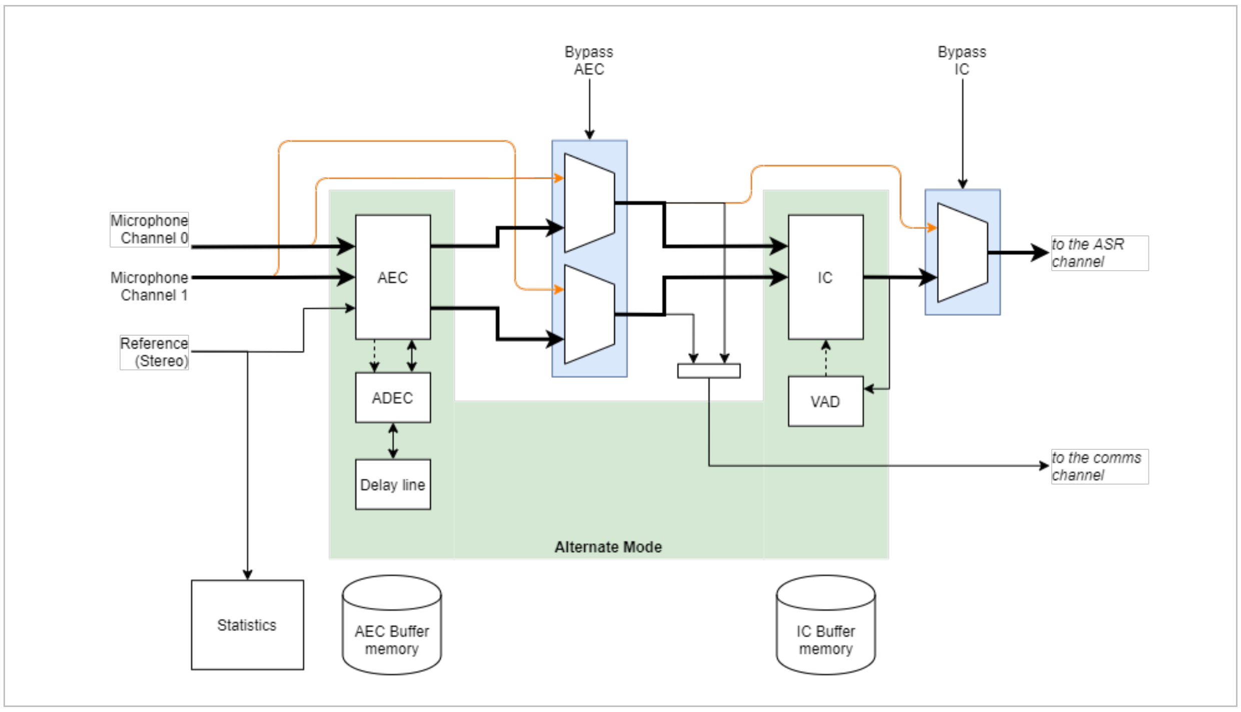


Figure 4-15 Audio pipeline configuration, [ALT_ARCH=0] mode

Whenever ALT_ARCH=1, then the pipeline dynamically switches between AEC alone, or IC alone. In this condition the AEC is able to make use of additional memory increasing the echo cancelling period, and making it more resilient to echo in highly reverberant conditions.

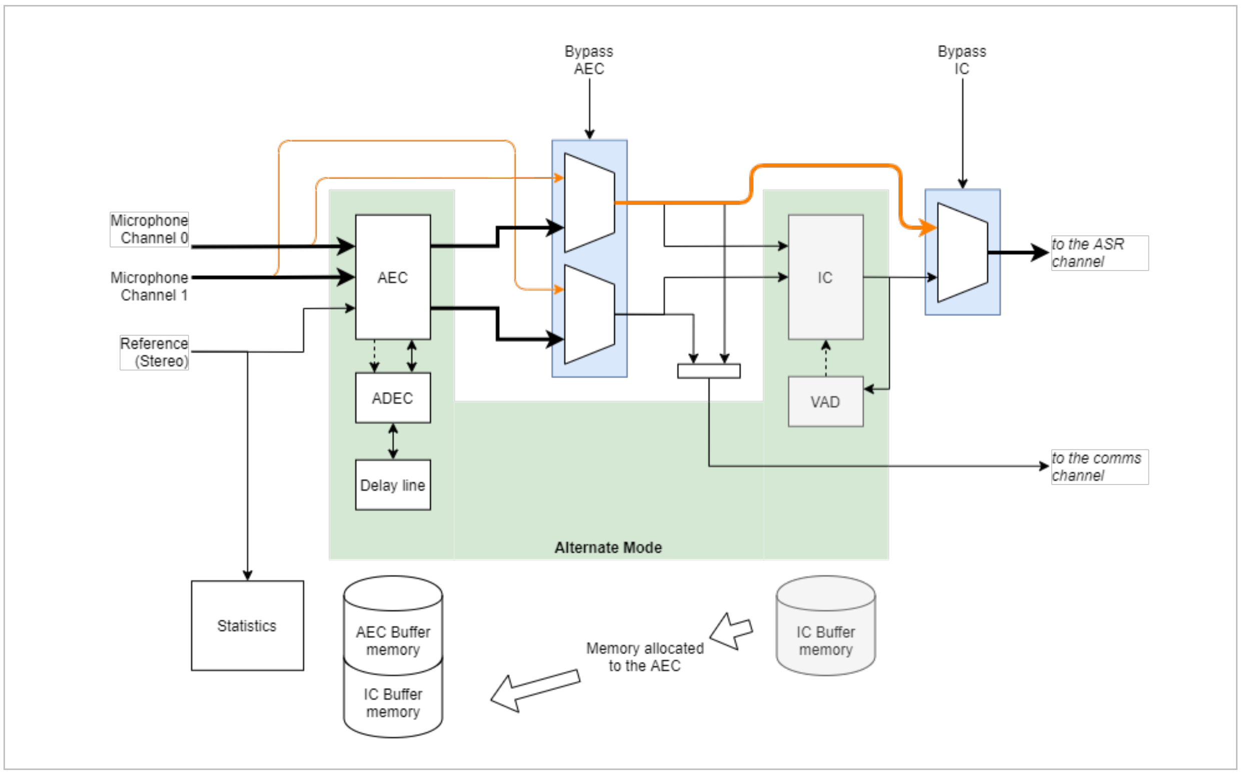


Figure 4-16 Audio pipeline configuration, [ALT_ARCH=1] when reference signal is present

The dynamic switching uses statistics collected from the reference signal to establish if echo cancelling is required.

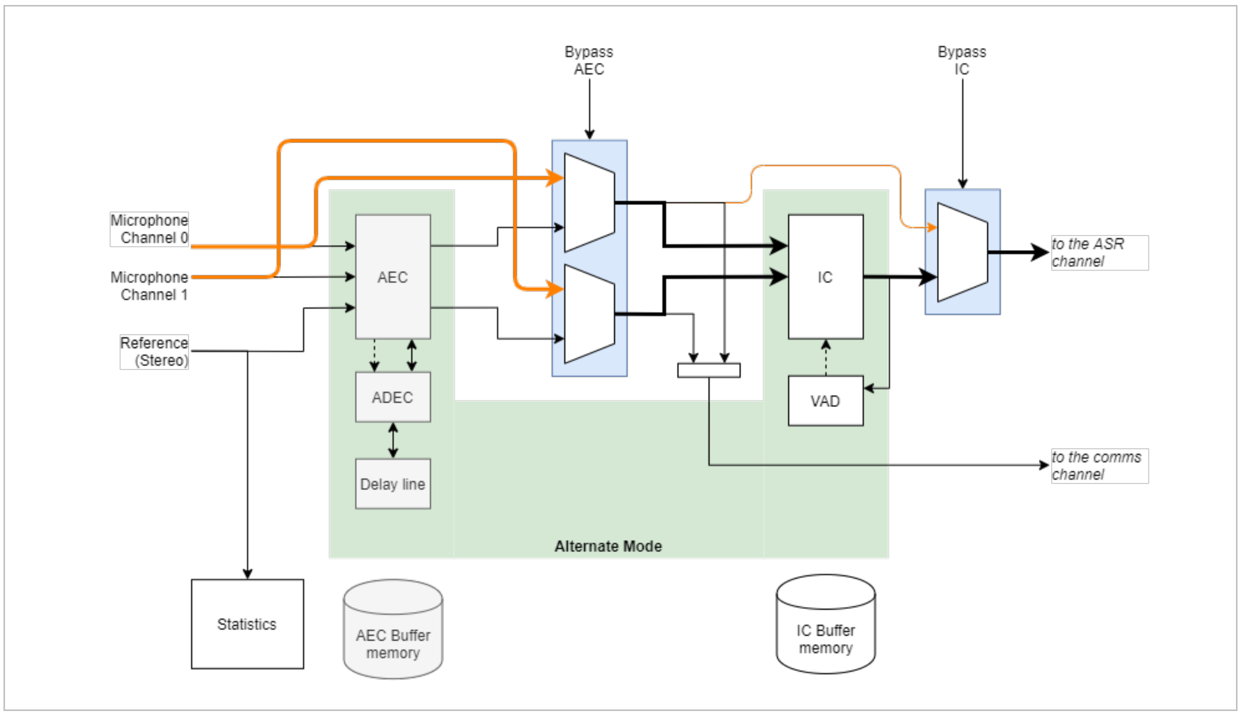


Figure 4-17 Audio pipeline configuration, [ALT_ARCH=1] when reference signal is absent

The following table summarises the audio characteristics for standard and alternative architectures.

Table 4-19 Alternative pipeline mode characteristics

PIPELINE CONFIGURATION	FAR-END AUDIO (AEC REF) STATUS	PIPELINE FUNCTIONALITY	AEC CHARACTERISTICS
ALT_ARCH = 0	With and without Far-end audio present	IC enabled AEC enabled	Max echo delay = 150mS
ALT_ARCH = 1	No far-end audio	IC enabled AEC disabled	No cancellation
ALT_ARCH = 1	Far-end audio present	IC disabled AEC enabled	Max echo delay = 225mS

The following table describes the configuration parameters for the Alternative Architecture.

Table 4-20 Alternative pipeline mode commands

COMMAND	TYPE	VALUE	DESCRIPTION
SET_ALT_ARCH_ENABLED GET_ALT_ARCH_ENABLED	uint32	[0,1] 0 - Alt arch is disabled. 1 - Alt arch is enabled (default)	Enable or disable alternate architecture (Alt arch). When alt arch is enabled, the system works in either AEC mode (when far end signal is detected) or IC mode (when far end signal is not detected). When in AEC mode in Alt arch, AEC processing happens on only one Mic channel with 15 phases per mic-ref AEC filter.

5. ADDITIONAL INFORMATION

5.1. DOCUMENTATION

Table 5-1 Additional documentation

DOCUMENT TITLE	DOWNLOAD
XVF3510-UA Datasheet	https://www.xmos.ai/file/xvf3510-ua-datasheet
XVF3510-INT Datasheet	https://www.xmos.ai/file/xvf3510-int-datasheet
XMOS xTIMEcomposer Tools User Guide	https://www.xmos.ai/file/tools-user-guide
XVF3510 Development Kit Setup Guides	https://www.xmos.ai/file/xvf3510-dev-kit-setup-guides

5.2. DEVICE FIRMWARE AND DRIVERS

Table 5-2 Device firmware

DEVICE FIRMWARE & APPLICATION SOFTWARE	DOWNLOAD
XVF3510-INT Firmware and Host applications	https://www.xmos.ai/file/xvf3510-int-release
XVF3510-UA Firmware and Host applications	https://www.xmos.ai/file/xvf3510-ua-release
xTIMEcomposer Programming Tools	https://www.xmos.ai/software-tools

6. REVISION HISTORY

DOCUMENT VERSION	RELEASE DATE	CHANGE DESCRIPTION
XM-014232-PC-2	23 Jul 2020	Describes V4.0 Firmware operation. Note: this document supersedes XM-013914-PC - XVF3510 control guide
XM-014232-PC-3	18 September 2020	XVF3510 V4.1 firmware release update: - Changes to SPI Slave boot process - Addition of USB volume control on -UA variant. Revised all links to point to xmos.ai website
XM-014232-PC-5	8 January 2021	Updated for V4.2 firmware: - New SPI boot process - Updates to AVS dev kit setup

APPENDICES

LIST OF APPENDICES

APPENDIX A: PARAMETER SUMMARY

The following section summarises the XVF3510 parameters which are programmable via the control interfaces or flash data partition. These parameters allow the setup of the XVF3510 processor's interfaces and tuning of the internal signal processing.

To aid quick reference of the key parameters the summary is split into two sections. The first details the most frequently used parameters which are required for interface configuration and basic control, and the second detail advanced parameters which will not generally need to be modified. Further details on the specific usage of parameters are discussed in the previous sections of the document and referenced below for convenience.

NOTE: The parameters shown below can be formatted into Read and Write commands, where appropriate by adding the prefix 'GET_' and 'SET_' for Read and Write respectively.

Table 6-1 Basic parameter summary ('-' used to indicate not applicable)

PARAMETER	READ / WRITE	DESCRIPTION	3510-UA DEFAULT	3510-INT DEFAULT
VERSION	R	Firmware version – See release notes and section 3.1 above	vX.Y.Z	vX.Y.Z
DELAY_SAMPLES	R/W	Configurable delay in samples	0	0
STATUS	R	Status	0	0
DEVICE_TO_USB_BIT_RES	R/W	Device to USB bit resolution	16	-
DEVICE_TO_USB_RATE	R/W	Device to USB rate	48000	-
GPI_INT_CONFIG	W	Sets the interrupt config for a specific pin	-	-
GPI_INT_PENDING_PIN	R	Read whether interrupt was triggered for selected pin	-	-
GPI_INT_PENDING_PORT	R	Read whether interrupt was triggered for all pins on selected port	-	-
GPI_PIN	R	Read current state of the selected GPIO pin	-	-
GPI_PIN_ACTIVE_LEVEL	W	Set the active level for a specific GPI pin. 0: active low 1: active high	0	0
GPI_PORT	R	Read current state of the selected GPIO port	-	-
GPI_READ_HEADER	R/W	Sets the selected port and pin for the next GPIO read	0 0	0 0
GPO_FLASHING	W	Set the serial flash mask for a specific pin. Each bit in the mask describes the GPO state for 100ms intervals	0	0
GPO_PIN	W	Write to a specific GPIO pin	-	-
GPO_PIN_ACTIVE_LEVEL	W	Set the active level for a specific GPO pin. 0: active low 1: active high	0	0
GPO_PORT	W	GPIO: Write to all pins of a GPIO port	-	-
GPO_PWM_DUTY	W	GPIO: Set the pwm duty for a specific pin. Value given as an integer percentage	0	0

PARAMETER	READ / WRITE	DESCRIPTION	3510-UA DEFAULT	3510-INT DEFAULT
I2S_RATE	R/W	I2S rate. This command can be used in SPI Boot Delay mode prior to SET_MIC_START_STATUS 1	48000	48000
I2S_START_STATUS	R/W	Start I2S. This command can be specified from the control interface in case of SPI booting INT device in delayed start mode.	-	-
IO_MAP	W	Set IO map for the device. arg1: dest arg2: source	-	-
IO_MAP_AND_SHIFT	R	Get IO map and output shift values for the device.	-	-
OUTPUT_SHIFT	W	For a selected output set the no. of bits the output samples will be shifted by. Positive shift value indicates left shift, negative indicates right shift.	-	-
SERIAL_NUMBER	R/W	Read / Write the serial number from USB descriptor (normally initialised from flash.	0	0
SYS_CLK_TO_MCLK_OUT_DIVIDER	R/W	Get XCore divider from system clock to output master clock. This command can be used in SPI Boot Delay mode prior to SET_MIC_START_STATUS 1	11	11
USB_BCD_DEVICE	R/W	USB Device Release Number (bcdDevice)	1	-
USB_PRODUCT_ID	R/W	USB Product ID	20 (0x0014)	-
USB_PRODUCT_STRING	R/W	Get USB Product string	XVF3510 (UAC1.0) Adaptive	-
USB_SERIAL_NUMBER	W	Load serial number from flash and initialise USB device descriptor with it. Will not work after boot since descriptor is populated only once with USB start.	-	-
USB_START_STATUS	R/W	Start USB. This command is only run from the flash. Run it only with -l option to generate the json item to use in the flash data-partition	-	0
USB_TO_DEVICE_BIT_RES	R/W	USB to device bit resolution	16	-
USB_TO_DEVICE_RATE	R/W	USB to device rate	48000	-
USB_VENDOR_ID	R/W	USB Vendor ID	8369 (0x20B1)	-
USB_VENDOR_STRING	R/W	USB Vendor string	XMOS	-
MCLK_IN_TO_PDM_CLK_DIVIDER	R/W	xCORE divider from input master clock to 6.144MHz DDR PDM microphone clock	2	2
ADEC_ENABLED	R/W	Automatic delay estimator controller enabled: 0: off 1: on	0	0
ADEC_MODE	R	Automatic delay estimator controller mode: 0: normal AEC mode 1: delay estimation mode	0	0

PARAMETER	READ / WRITE	DESCRIPTION	3510-UA DEFAULT	3510-INT DEFAULT
DELAY_DIRECTION	R/W	Configurable delay direction: 0: delay references 1: delay mics	0	0
DELAY_ESTIMATE	R	Delay estimate	-	-
DELAY_ESTIMATOR_ ENABLED	R/W	Enable/disable delay estimation	0	0
MANUAL_ADEC_CYC LE_TRIGGER	W	Trigger a delay estimate	-	-
MIC_SHIFT_SATURAT E	R/W	The shift value and saturation (1=enable) to be applied to the input mic samples	0 0	0 0

Table 6-2 Advanced parameter summary

PARAMETER	READ / WRITE	DESCRIPTION	3510-UA DEFAULT	3510-INT DEFAULT
BLD_HOST	R	Build host	Jenkins	Jenkins
BLD_MODIFIED	R	Build modified from given view/hash	false	false
BLD_MSG	R	Build message	Default	Default
BLD_REPO_HASH	R	Repo hash – unique source version	See release notes	See release notes
BLD_XGIT_HASH	R	xgit hash – unique build version	See release notes	See release notes
BLD_XGIT_VIEW	R	xgit view	sw_xvf3510_master	sw_xvf3510_master
FILTER_BYPASS	R/W	Filter bypass state. arg1: 0 - filter enabled 1 - bypassed	1	1
FILTER_COEFF	R/W	Set biquad coeffs for a selected filter using floating point. arg1..10: 5x2 float coeffs in forward order (a1, a2, b0, b1, b2) where a0 always is 1.0.	0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000	0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
FILTER_COEFF_RAW	R/W	Set raw biquad coeffs for a selected filters. arg1..10: 2 sets of coeffs in forward order (b0, b1, b2, =-E1, -a2) signed Q28 format	0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0
FILTER_INDEX	R/W	Set filter index. Selects which filter block will be read from/written to arg1: dest	0	0
HARDWARE_BUILD	R	Get the build number from the hardware build section of the flash data partition.	-1	-1
I2C	R/W	Read from an I2C device connected to the xvf device	-	-
I2C_READ_HEADER	R/W	Get the address register address and count of next I2C read	0 0 0	-

PARAMETER	READ / WRITE	DESCRIPTION	3510-UA DEFAULT	3510-INT DEFAULT
I2C_WITH_REG	R/W	Read from the register of an I2C device connected to the xvf device	-	-
MONITOR_STATE_USING_GPO_ENABLED	W	Enable monitoring of state on GPO. This command is only run from the flash. Run it only with -l option to generate the json item to use in the flash data-partition	-	-
KWD_BOOT_STATUS	R	Gets boot status for keyword detectors	0	0
KWD_INTERRUPT_PIN	R/W	GPI pin index to receive keyword interrupt on	4	4
MAX_UBM_CYCLES	R	Get maximum no. of cycles taken by the user buffer management function	-	-
MIC_START_STATUS	R/W	Get microphone client start status.	2	2
REMAX_UBM_CYCLES	W	Reset the max user buffer management cycles count	-	-
RUN_STATUS	R	Gets run status for the device (See Appendix B)	-	-
SPI	R	Gets the contents of the SPI read buffer	-	-
SPI_PUSH	W	Push SPI command data onto the execution queue	-	-
SPI_PUSH_AND_EXEC	W	Push SPI command data and execute the command from the stack	-	-
SPI_READ_HEADER	R/W	Address and count of next SPI read	0 0	0 0
ADAPTATION_CONFIG_AEC	R/W	Adaptation config 0 = filter adapt with variable stepsize 1 = filter adapt with fixed stepsize 2 = filter fixed	0	0
BYPASS_AEC	R/W	AEC bypass	1	1
COEFF_INDEX_AEC	R/W	AEC coefficient index	0	0
ERLE_CH0_AEC	R	AEC channel 0 ERLE	-	-
ERLE_CH1_AEC	R	AEC channel 1 ERLE	-	-
F_BIN_COUNT_AEC	R	AEC f bin count	257	257
FILTER_COEFFICIENTS_AEC	R	AEC filter coefficients	-	-
FORCED_MU_VALUE_AEC	R/W	AEC forced mu value	1	1
FRAME_ADVANCE_AEC	R	AEC frame advance	240	240
MU_LIMITS_AEC	R/W	AEC mu_high and mu_low	1.0000 0.0001	1.0000 0.0001
MU_SCALAR_AEC	R/W	AEC get mu_scalar	0.4	0.4

PARAMETER	READ / WRITE	DESCRIPTION	3510-UA DEFAULT	3510-INT DEFAULT
RESET_FILTER_AEC	W	AEC reset filter. Note: do NOT prefix with SET_	-	-
SIGMA_ALPHAS_AEC	R/W	AEC sigma alphas	5 5 11	5 5 11
X_CHANNEL_PHASES_AEC	R	AEC x channel phases	15 15 4 0 0 0 0 0 0 0	15 15 4 0 0 0 0 0 0 0
X_CHANNELS_AEC	R	AEC x channels	2	2
X_ENERGY_DELTA_AEC	R/W	AEC X energy delta	-	-
X_ENERGY_GAMMA_LOG2_AEC	R/W	AEC X energy gamma log2	-	-
Y_CHANNELS_AEC	R	AEC y channels	1	1
ADAPTATION_CONFIG_IC	R/W	IC: get adaptation config	0	0
BYPASS_IC	R/W	IC: get bypass state	0	0
CH1_BEAMFORM_ENABLE	R/W	Channel 1 Beamforming enabled	1	1
COEFFICIENT_INDEX_IC	R/W	IC Coefficient index	0	0
FILTER_COEFFICIENTS_IC	R	IC Filter coefficients	-	-
FORCED_MU_VALUE_IC	R/W	IC forced mu value	-	-
PHASES_IC	R	IC phases	10	10
PROC_FRAME_BINS_IC	R	IC proc frame bins	256	256
RESET_FILTER_IC	W	IC reset filter, note: do not prefix with SET_	-	-
SIGMA_ALPHA_IC	R/W	IC adaptation config	11	11
X_ENERGY_DELTA_IC	R/W	IC X energy delta	-	-
X_ENERGY_GAMMA_LOG2_IC	R/W	IC X energy gamma log2	-	-
BYPASS_SUP	R/W	SUP bypass	0	0
ENABLED_AES	R/W	SUP echo suppression enabled (DO NOT ENABLE)	0	0
ENABLED_NS	R/W	SUP noise suppression enabled	1	1
NOISE_FLOOR_NS	R/W	SUP noise suppression noise floor	0.1259	0.1259
ADEC_FAR_THRESHOLD	R/W	ADEC Far-end signal energy threshold above which AGM is updated	0.000002	0.000002

PARAMETER	READ / WRITE	DESCRIPTION	3510-UA DEFAULT	3510-INT DEFAULT
ADEC_PEAK_TO_AVERAGE_GOOD_AEC	R/W	ADEC the peak to average ratio that is considered good when in normal AEC mode	4.000000	4.000000
ADEC_TIME_SINCE_RESET	R	Time in milliseconds since last automatic delay change by ADEC	-	-
AEC_PEAK_TO_AVERAGE_RATIO	R	AEC coefficients peak to average ratio	-	-
AGM	R	AEC Goodness Metric estimate (0.0 - 1.0)	-	-
ALT_ARCH_ENABLED	R/W	State of XVF3510 alternate architecture setting	1	1
ERLE_BAD_BITS	R/W	ERLE bad threshold in bits (log2)	-	-
ERLE_BAD_GAIN	R/W	Set how steeply AGM drops off when ERLE below threshold	0.0664	0.0664
ERLE_GOOD_BITS	R/W	ERLE good threshold in bits (log2)	2	2
LOCKER_DELAY_SETPOINT_DIRECTION	R/W	Delay set point direction	0	0
LOCKER_DELAY_SETPOINT_ENABLED	R/W	Delay set point enabled	0	0
LOCKER_DELAY_SETPOINT_SAMPLES	R/W	Delay setpoint samples	0	0
LOCKER_ENABLED	R/W	Locker delay detection and control	0	0
LOCKER_NUM_BAD_FRAMES_THRESHOLD	R/W	No. of bad peak to avg ERLE frames that locker sees before it triggers ADEC.	666	666
LOCKER_STATE	R	Locker state	BOTH_WAIT	BOTH_WAIT
MAX_CONTROL_TIME_STAGE_A	R	Max control time per frame	-	-
MAX_DSP_TIME_STAGE_A	R	Max dsp time per frame	-	-
MAX_IDLE_TIME_STAGE_A	R	Max idle time per frame	-	-
MAX_RX_TIME_STAGE_A	R	Max rx time per frame	-	-
MAX_TX_TIME_STAGE_A	R	Max tx time per frame	-	-
MIN_CONTROL_TIME_STAGE_A	R	Min control time per frame	-	-
MIN_DSP_TIME_STAGE_A	R	Min dsp time per frame	-	-
MIN_IDLE_TIME_STAGE_A	R	Min idle time per frame	-	-

PARAMETER	READ / WRITE	DESCRIPTION	3510-UA DEFAULT	3510-INT DEFAULT
MIN_RX_TIME_STAGE_A	R	Min rx time per frame	-	-
MIN_TX_TIME_STAGE_A	R	Min tx time per frame	-	-
PEAK_PHASE_ENERGY_TREND_GAIN	R/W	Value which sets AGM sensitivity to peak phase energy slope	3	3
PHASE_POWER_INDEX	R/W	ERLE gain	0	0
PHASE_POWERS	R	5 phase powers (240 samples per phase) used in delay estimation from the index set.	0.000000 dB 0.000000 dB 0.000000 dB 0.000000 dB 0.000000 dB	0.000000 dB 0.000000 dB 0.000000 dB 0.000000 dB 0.000000 dB
RESET_TIME_STAGE_A	W	Reset stage A frame time	-	-
MAX_CONTROL_TIME_STAGE_B	R	Max control time per frame	-	-
MAX_DSP_TIME_STAGE_B	R	Stage B max dsp time per frame	-	-
MAX_IDLE_TIME_STAGE_B	R	Stage B max idle time per frame	-	-
MAX_RX_TIME_STAGE_B	R	Stage B max rx time per frame	-	-
MAX_TX_TIME_STAGE_B	R	Stage B max tx time per frame	-	-
MIN_CONTROL_TIME_STAGE_B	R	Stage B min control time per frame	-	-
MIN_DSP_TIME_STAGE_B	R	Stage B min dsp time per frame	-	-
MIN_IDLE_TIME_STAGE_B	R	Stage B min idle time per frame	-	-
MIN_RX_TIME_STAGE_B	R	Stage B min rx time per frame	-	-
MIN_TX_TIME_STAGE_B	R	Stage B min tx time per frame	-	-
RESET_TIME_STAGE_B	W	Reset stage B frame time	-	-
ADAPT_CH0_AGC	R/W	AGC adaptation for channel 0	1	1
ADAPT_CH1_AGC	R/W	AGC adaptation for channel 1	1	1
DECREMENT_GAIN_STEPSIZE_CH0_AGC	R/W	Stepsize with which gain is decremented for AGC ch0	0.87	0.87
DECREMENT_GAIN_STEPSIZE_CH1_AGC	R/W	Stepsize with which gain is decremented for AGC ch1	0.988	0.988
GAIN_CH0_AGC	R/W	Gain for channel 0	-	-
GAIN_CH1_AGC	R/W	Gain for channel 1	-	-

PARAMETER	READ / WRITE	DESCRIPTION	3510-UA DEFAULT	3510-INT DEFAULT
INCREMENT_GAIN_STEPSIZE_CH0_AGC	R/W	Stepsize with which gain is incremented for AGC ch0	1.197	1.197
INCREMENT_GAIN_STEPSIZE_CH1_AGC	R/W	Stepsize with which gain is incremented for AGC ch1	1.0034	1.0034
LC_ENABLED_CH0_AGC	R/W	Loss control enable for channel 0	0	0
LC_ENABLED_CH1_AGC	R/W	Loss control enable for channel 1	1	1
LOWER_THRESHOLD_CH0_AGC	R/W	Lower threshold of AGC desired level for channel 0	0.1905	0.1905
LOWER_THRESHOLD_CH1_AGC	R/W	Lower threshold of AGC desired level for channel 1	0.4	0.4
UPPER_THRESHOLD_CH0_AGC	R/W	Upper threshold of AGC desired level for channel 0	0.7079	0.7079
UPPER_THRESHOLD_CH1_AGC	R/W	Upper threshold of AGC desired level for channel 1	0.4	0.4
MAX_CONTROL_TIME_STAGE_C	R	Stage C max control time per frame	-	-
MAX_DSP_TIME_STAGE_C	R	Stage C max dsp time per frame	-	-
MAX_GAIN_CH0_AGC	R/W	Max gain for channel 0	999.9847	999.9847
MAX_GAIN_CH1_AGC	R/W	Max gain for channel 1	999.9847	999.9847
MAX_IDLE_TIME_STAGE_C	R	Stage C max idle time per frame	-	-
MAX_RX_TIME_STAGE_C	R	Stage C max rx time per frame	-	-
MAX_TX_TIME_STAGE_C	R	Stage C max tx time per frame	-	-
MIN_CONTROL_TIME_STAGE_C	R	Stage C min control time per frame	-	-
MIN_DSP_TIME_STAGE_C	R	Stage C min dsp time per frame	-	-
MIN_IDLE_TIME_STAGE_C	R	Stage C min idle time per frame	-	-
MIN_RX_TIME_STAGE_C	R	Stage C min rx time per frame	-	-
MIN_TX_TIME_STAGE_C	R	Stage C min tx time per frame	-	-
REF_OUT_CH1	R/W	Stage C: check if reference audio is output in channel 1	0	0
RESET_TIME_STAGE_C	W	Reset stage C frame time	-	-

APPENDIX B: BOOT STATUS CODES (RUN_STATUS)

The following table describes the Boot status codes returned by the startup processes accessible though the GET_RUN_STATUS control utility command.

CODE	LABEL	NOTE
0	INIT	Reserved initial value. Decline attempts to initiate DFU.
1	DATA_PARTITION_NOT_FOUND	Not used.
2	FACTORY_DATA_SUCCESS	Normal operation.
3	UPGRADE_DATA_SUCCESS	Normal operation.
4	FACTORY_DATA_IN_PROGRESS	Image scanning in progress. Decline attempts to initiate DFU.
5	UPGRADE_DATA_IN_PROGRESS	Image scanning in progress. Decline attempts to initiate DFU.
6	DFU_IN_PROGRESS	Enough DFU commands received to establish a connection to on-board flash memory. Not cleared until reboot.
7	HW_BUILD_READ_SUCCESS	Reserved intermediate value. Normally never returned.
8	HW_BUILD_PARTITION_SIZE_ERROR	Problem reading data partition header. Check factory programming.
9	HW_BUILD_PARTITION_BASE_ERROR	Problem reading data partition header. Check factory programming.
10	HW_BUILD_READ_ERROR	Problem reading data partition header. Check factory programming.
11	HW_BUILD_CRC_ERROR	Problem reading data partition header. Check factory programming. May indicate that no data partition is present or a flash wear issue.
12	HW_BUILD_TAG_ERROR	Problem reading data partition header. Check factory programming.
13	FACTORY_VERSION_ERROR	No valid upgrade image found. A factory image did not match running version. This can indicate fail-safe mode.
14	UPGRADE_VERSION_ERROR	Valid upgrade boot and data images found but data image version does not match running version. Check correct version of deployed field upgrade.
15	FACTORY_ITEM_READ_ERROR	Problem reading configuration items from data image. Unexpected error.
16	UPGRADE_ITEM_READ_ERROR	Problem reading configuration items from data image. Unexpected error.
17	FACTORY_ITEM_INVALID_TYPE	Last item encountered is not of terminator type. Should never happen with script generated data images. Check generation procedure.
18	UPGRADE_ITEM_INVALID_TYPE	Last item encountered is not of terminator type. Should never happen with script generated data images. Check generation procedure.
19	DFU_FLASH_CONNECT_FAILED	Failed to establish on-board flash connection. Check factory programming. Check flash specification (see section below).
20	DFU_FLASH_SPEC_UNSUITABLE	Flash specification unsuitable for DFU. Check flash specification (see section below).

APPENDIX C: EXAMPLE .SPISPEC FILE FORMAT

SPISPEC file for Adesto AT25SF161.

```
    0,                          /* AT25SF161 - Just specify 0 as flash_id */
    256,                        /* page size */
    8192,                       /* num pages */
    3,                          /* address size */
    4,                          /* log2 clock divider */
    0x9F,                       /* QSPI_RDID */
    0,                          /* id dummy bytes */
    3,                          /* id size in bytes */
    0,                          /* device id (leave zero) */
    0x20,                       /* QSPI_SE */
    4096,                       /* Sector erase is always 4KB */
    0x06,                       /* QSPI_WREN */
    0x04,                       /* QSPI_WRDI */
    PROT_TYPE_SR,               /* Protection via SR */
    {{0x18,0x00},{0,0}},        /* QSPI_SP, QSPI_SU */
    0x02,                       /* QSPI_PP */
    0xEB,                       /* QSPI_READ_FAST */
    1,                          /* 1 read dummy byte */
    SECTOR_LAYOUT_REGULAR,      /* mad sectors */
    {4096,{0,{0}}},             /* regular sector sizes */
    0x05,                       /* QSPI_RDSR */
    0x01,                       /* QSPI_WRSR */
    0x01,                       /* QSPI_WIP_BIT_MASK */
```

APPENDIX D: SPI BOOT CUSTOM CONNECTION

SPI BOOT UA XVF3510 DEVICE

The *UA* release package contains the `send_image_from_rpi.py` and the SPI bootable image of the corresponding build in the `bin` folder. If booting an UA binary via SPI, the ribbon cable between the PiHat and the XVF3510 device must be disconnected and the following pins must be connected:

SIGNAL CONNECTION (ODD PINS)	CONNECTOR (ODD PINS)	CONNECTOR (EVEN PINS)	SIGNAL CONNECTION (EVEN PINS)
	1	2	
	3	4	
	5	6	GND*
	7	8	BOOT_SEL
GND*	9	10	RST_N
	11	12	
	13	14	GND*
	15	16	
	17	18	
SPI_MOSI	19	20	GND*
SPI_MISO	21	22	
SPI_CLK	23	24	SPI_CSn
GND*	25	26	
	27	28	
	29	30	GND*
	31	32	
	33	34	GND*
	35	36	
	37	38	
GND*	39	40	

* Note: all ground connections need to be connected.

APPENDIX E: USB ENUMERATION

The XVF3510 includes a Human Interface Device (HID) endpoint to enable the XVF3510 to signal interrupts caused by GPIO events. The table below shows how the XVF3510 HID appears on Windows using USB view.

DEVICE NAME	DESCRIPTION	DEVICE TYPE	VENDOR ID	PRODUCT ID	USB CLASS	USB SUBCLASS	USB PROTOCOL	SERVICE NAME	USB VERSION	DRIVER DESCRIPTION
XVF3510 (UAC1.0) Adaptive	USB Composite Device	Unknown	20b1	0014	00	00	00	usbccgp	2.00	USB Composite Device
XVF3510 (UAC1.0) Adaptive	USB Audio Device	Audio	20b1	0014	01	01	00	usbaudio	2.00	USB Audio Device
XVF3510 (UAC1.0) Adaptive	XMOS Control	Vendor Specific	20b1	0014	ff	ff	ff		2.00	
XVF3510 (UAC1.0) Adaptive	USB Input Device	HID (Human Interface Device)	20b1	0014	03	00	00	HidUsb	2.00	USB Input Device

During USB enumeration, the XVF3510 HID produces three descriptors. The listing below shows them as recorded on Windows using USB View. For details of the structure and meaning of these descriptors, see the *USB Specification v2.0* sections 9.6.5 and 9.6.6 and the *Device Class Definition for Human Interface Devices (HID) v1.11* section 6.2.1.

```
          ===>Interface Descriptor<===
bLength:                           0x09
bDescriptorType:                   0x04
bInterfaceNumber:                  0x04
bAlternateSetting:                 0x00
bNumEndpoints:                     0x01
bInterfaceClass:                   0x03  -> HID Interface Class
bInterfaceSubClass:                0x00
bInterfaceProtocol:                0x00
iInterface:                        0x00
          ===>HID Descriptor<===
bLength:                            0x09
bDescriptorType:                    0x21
bcdHID:                           0x0110
bCountryCode:                       0x00
bNumDescriptors:                    0x01
bDescriptorType:                    0x22 (Report Descriptor)
wDescriptorLength:                0x002B
         ===>Endpoint Descriptor<===
bLength:                            0x07
bDescriptorType:                    0x05
bEndpointAddress:                   0x82  -> Direction: IN - EndpointID: 2
bmAttributes:                       0x03  -> Interrupt Transfer Type
wMaxPacketSize:                   0x0040 = 0x40 bytes
bInterval:                          0x08
```

APPENDIX F: USB HID - EXAMPLE USING THE DEVELOPMENT KIT

WORKED EXAMPLE

An XVF3510 development kit, a Raspberry Pi and a jump wire are required for this example.

The development kit should be configured as XVF3510-UA. Instructions on updating the firmware are available in the *Updating the firmware* section.

The development kit should then be connected to a Raspberry Pi and set up according to the development kit setup guide. Extract the Raspberry Pi host utilities from the release package, and use them to enable interrupts like so:

```
vfctrl_usb.exe SET_GPI_INT_CONFIG 0 0 3
```

The HID events can be observed on `/dev/input/event0` on the Raspberry Pi either directly (eg `xxd`) or using the `evtest` utility (normally available through APT on Raspbian).

`event0` will be the correct HID device is most cases. If the test system has additional sources of events, the correct one can be identified under `/dev/input` by looking at the `Handlers` line in the output of `/proc/bus/input/devices`.

Now toggle INT_N signal on the XK-VF3510 board by connecting it to 3V3 and GND using a jump wire. Example output from `evtest` is:

```
Input driver version is 1.0.1
Input device ID: bus 0x3 vendor 0x20b1 product 0x14 version 0x110
Input device name: "XMOS XVF3510 (UAC1.0) Adaptive"
Supported events:
  Event type 0 (EV_SYN)
  Event type 1 (EV_KEY)
    Event code 128 (KEY_STOP)
    Event code 193 (KEY_F23)
    Event code 194 (KEY_F24)
    Event code 217 (KEY_SEARCH)
  Event type 4 (EV_MSC)
    Event code 4 (MSC_SCAN)
Key repeat handling:
  Repeat type 20 (EV_REP)
    Repeat code 0 (REP_DELAY)
      Value    250
    Repeat code 1 (REP_PERIOD)
      Value     33
Properties:
Testing ... (interrupt to exit)
Event: time 1586524983.094859, type 4 (EV_MSC), code 4 (MSC_SCAN), value 70072
Event: time 1586524983.094859, type 1 (EV_KEY), code 193 (KEY_F23), value 1
Event: time 1586524983.094859, -------------- SYN_REPORT ------------
Event: time 1586524983.353655, type 1 (EV_KEY), code 193 (KEY_F23), value 2
Event: time 1586524983.353655, -------------- SYN_REPORT ------------
Event: time 1586524983.403671, type 1 (EV_KEY), code 193 (KEY_F23), value 2
Event: time 1586524983.403671, -------------- SYN_REPORT ------------
Event: time 1586524983.453659, type 1 (EV_KEY), code 193 (KEY_F23), value 2
Event: time 1586524983.453659, -------------- SYN_REPORT ------------
```

APPENDIX G: GENERAL PURPOSE FILTER EXAMPLE

WORKED EXAMPLE

Steps in this example:

- Set the stereo USB output to listen to the stereo USB input (loopback, skipping audio processing pipeline completely)
- Apply a stereo 500Hz high-pass and 4kHz low-pass cascaded biquad filter
  - The 500Hz high-pass filter coefficients are:
- a1 = -1.90748889
  a2 = 0.91158173
  b0 = 0.95476766
  b1 = -1.90953531
  b2 = 0.95476766
- The 4kHz low-pass filter coefficients are:
- a1 = -1.27958194
  a2 = 0.47753396
  b0 = 0.04948800
  b1 = 0.09897601
  b2 = 0.04948800
- Hear the effect filtered signals when the filters are enabled

This example assumes that the input and output sample rate is 48kHz.

First, connect the USB output to the USB input:

```
vfctrl_usb SET_IO_MAP 0 7 # (USB output left outputs USB input left)
vfctrl_usb SET_IO_MAP 1 8 # (As above for right channel)
```

Now configure the filter:

```
vfctrl_usb SET_FILTER_INDEX 2 (USB output left filter)
vfctrl_usb SET_FILTER_COEFF -1.90748889 0.91158173 0.95476766 -1.90953531
0.95476766 -1.27958194 0.47753396 0.04948800 0.09897601 0.04948800
vfctrl_usb SET_FILTER_INDEX 3 (USB output right filter)
vfctrl_usb SET_FILTER_COEFF -1.90748889 0.91158173 0.95476766 -1.90953531
0.95476766 -1.27958194 0.47753396 0.04948800 0.09897601 0.04948800
```

Now enable the filter:

```
vfctrl_usb SET_FILTER_INDEX 0
vfctrl_usb SET_FILTER_BYPASS 0
vfctrl_usb SET_FILTER_INDEX 1
vfctrl_usb SET_FILTER_BYPASS 0
```

Play a white noise source from the USB device and record the input. Use a spectrogram to show the band limited signal due to the effect of the filters. The effect should also be audible.

APPENDIX H: COMMAND TRANSPORT PROTOCOL

TRANSPORTING CONTROL PARAMETERS PROTOCOL

Control parameters are converted to an array of bytes in network byte order (big endian) before they're sent over the transport protocol. For example, to set a control parameter to integer value 305419896 which corresponds to hex 0x12345678, the array of bytes sent over the transport protocol would be {0x12, 0x34, 0x56, 0x78}. Similarly, a 4 byte payload {0x00, 0x01, 0x23, 0x22} read over the transport protocol is interpreted as an integer value 0x00012322.

In addition to the control parameters values, commands include Resource ID, the Command ID and Payload Length fields that must be communicated from the host to the device. The Resource ID is an 8-bit identifier that identifies the resource within the device that the command is for. The Command ID is an 8-bit identifier used to identify a command for a resource in the device. Payload length is the length of the data in bytes that the host wants to write to the device or read from the device.

The payload length is interpreted differently for GET_ and SET_ commands. For SET_commands, the payload length is simply the number of bytes worth of control parameters to write to the device. For example, the payload length for a SET_ command to set a control parameter of type int32 to a certain value, would be set to 4. For GET_ commands the payload length is 1 more than the number of bytes of control parameters to read from the device. For example, a GET_ command to read a parameter of type int32, payload length would be set to 5. The one extra byte is used for status and is the first byte (payload[0]) of the payload received from the device. In the example above, payload[0] would be the status byte and payload[1]..payload[4] would be the 4 bytes that make up the value of the control parameter.

The table below lists the different values of the status byte and the action the user is expected to take for each status:

RETURN CODE	VALUE	DESCRIPTION
ctrl_done	0	Read command successful. The payload bytes contain valid payload returned from the device
ctrl_wait	1	Read command not serviced. Retry until ctrl_done status returned
ctrl_invalid	3	Error in read command. Abort and debug

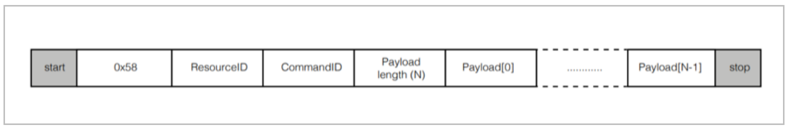
The GET_commands need the extra status byte since the device might not return the control parameter value immediately due to timing constraints. If that is the case the status byte would indicate the status as ctrl_wait and the user would need to retry the command. When returned a ctrl_wait, the user is expected to retry the GET_ command until the status is returned as ctrl_done. The first GET_command is placed in a queue and it will be serviced by the end of each 15ms audio frame. Once the status byte indicates ctrl_done, the rest of the bytes in the payload indicate the control parameter value.

TRANSPORTING CONTROL PARAMETERS OVER I2C

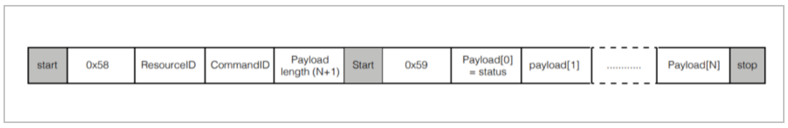
This section describes the I2C command sequence when issuing read and write commands to the device.

The first byte sent over I2C after start contains the device address and information about whether this is an I2C read transaction or a write transaction. This byte is 0x58 for a write command or 0x59 for a read command. These values are derived by left shifting the device address (0x2c) by 1 and doing a logical OR of the resulting value with 0 for an I2C write and 1 for an I2C read.

The bytes sequence sent between I2C start and stop for SET_ commands is shown in the figure below.



For GET_ commands, the I2C commands sequence consists of a write command followed by a read command with a repeated start between the 2 commands. The write command writes the resource ID, command ID and the expected data length to the device and the read command reads the status byte followed by the rest of the payload that makes up the control parameter value. The figure below shows the I2C bytes sequence sent and received for a GET_ command.



TRANSPORTING CONTROL PARAMETERS OVER USB

Use the vendor_id 0x20b1, product_id 0x0014 and interface number 3 to initialize for USB. The API function libusb_control_transfer() is used for transporting over USB. When calling libusb_control_transfer(), wIndex corresponds to the Resource ID, wValue is the Command ID and wLength is the payload length.

FLOATING POINT TO FIXED POINT (Q FORMAT) CONVERSION

Numbers with fractional parts can be represented as floating-point or fixed-point numbers. Floating point formats are widely used but carry performance overheads. Fixed point formats can improve system efficiency and are used extensively within the XVF3510. Fixed point numbers have the position of the decimal point fixed and this is indicated as a part of the format description.

In this document, Q format is used to describe fixed point number formats, with the representation given as Q$m.n$ format where m is the number of bits reserved for the sign and integer part of the number and n is the number of bits reserved for the fractional part of the number. The position of the decimal point is a trade-off between the range of values supported and the resolution provided by the fractional bits.

The dynamic range of Q$m.n$ format is -2^{m-1} and $2^{m-1}-2^{-n}$ with a resolution of 2^{-n}

To convert a floating-point format number to Q$m.n$ format fixed-point number:

- Multiply the floating-point number by 2^m.
- Round the result to the nearest integer.
- The resulting integer number is the Q$m.n$ fixed-point representation of the initial floating-point number.

To convert a Q$m.n$ fixed-point number to floating-point:

- Divide the fixed-point number by 2^m.
- The resulting decimal number is a floating-point representation of the fixed-point number.

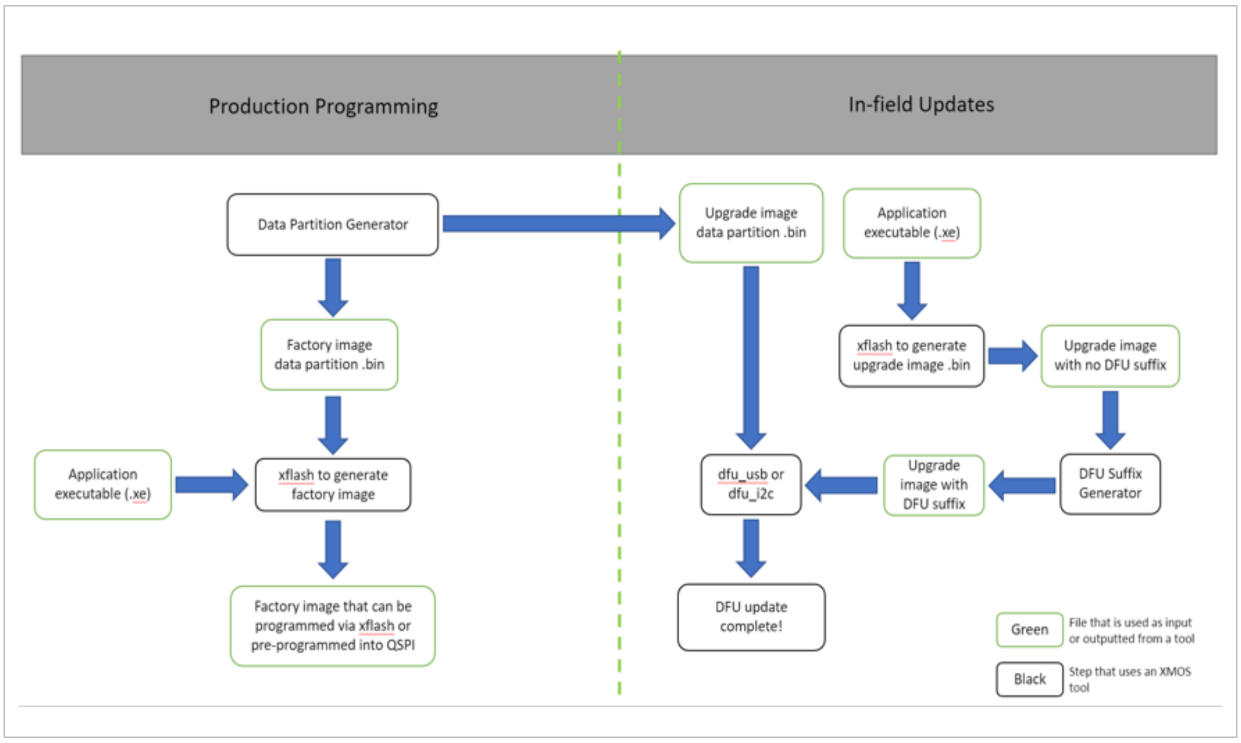
Converting a number into fixed point format and then back to a floating point number may introduce an error of up to $\pm 2^{-(n+1)}$

Example:

To represent a floating-point number 14.765467 in Q8.24 format, the equivalent fixed-point number would be 14.765467 x 2^{24} = 247723429.2 which rounds to 247723429.

To get back the floating-point number given the Q8.24 number 247723429, calculate 247723429 ÷ 2^{24} and get back the floating-point number as 14.76546699. The difference of 0.00000001 is correct to with the error bounds of $\pm 2^{-25}$ which is ±0.00000003

APPENDIX I: FLASH PROGRAMMING AND UPDATE FLOW



Production Programming
In-field Updates
Data Partition Generator
Factory image data partition .bin
Application executable (.xe)
xflash to generate factory image
Factory image that can be programmed via xflash or pre-programmed into QSPI
Upgrade image data partition .bin
Application executable (.xe)
xflash to generate upgrade image .bin
Upgrade image with no DFU suffix
DFU Suffix Generator
Upgrade image with DFU suffix
dfu_usb or dfu_i2c
DFU update complete!
Green
File that is used as input or outputted from a tool
Black
Step that uses an XMOS tool

APPENDIX J: CAPTURING PACKED SAMPLES FOR SYSTEM INTEGRATION

To assist with system integration, the XVF3510 provides the ability to pack multiple 16kHz channels into a 48kHz output. The following section describes the usage of packed signals.

All packed functions provide a **snapshot of a 16kHz signals over a 48kHz output**. If the output stream is not 48kHz, it will not work because the 3x bandwidth is needed for packing the 16kHz signals. They all also require that **no volume scaling be applied on the host** otherwise it will break the marker sequence resulting in the captured audio being unable to be unpacked. There are two packing mechanisms however for typical usage where a full capture of the pipeline is needed, PACKED_ALL is recommended.

CAPTURING PACKED_ALL SIGNALS

`PACKED_ALL` packs 3 channels into a single channel (mic, ref, pipeline out) so requires two output channels to capture all 6 signals of interest. When using -UA it uses the bit resolution of the USB output interface (even if you output to I2S on the -UA device) or assumes 32b it you are using -INT where the output interface is always I2S. The sequence is as follows:

- top bitres-1 bits of mic sample with LSbit marker ‘0’
- top bitres-1 bits of ref sample with LSbit marker ‘1’
- top bitres-1 bits of pipeline oit sample with LSbit marker ‘1’

The `unpacker_packed_all.py`script looks for 0, 0, 1 for the LS bit to check for a packed_all sequence, else it will report an error. This packing will work with 16b, 24b and 32b USB bit width. It is not bit-perfect (it loses 1b for the marker). It can work on a Mac if you use a 16b or 24b output resolution on -UA device. Since microphones signal levels are quite low from the output of the decimators, it is recommended to use at least 24b resolution to keep the quantisation noise floor down with respect to signal.

CAPTURING ALL PIPELINE INPUT AND OUTPUT SIGNALS OVER A 48KHZ USB INTERFACE

The goal here is to capture the pipeline input and output to provide visibility on what signals are actually entering the pipeline and what processed output was generated. This can be useful when checking the microphone and reference signals are correctly routed, as well as checking signal delay issues causing poor AEC performance.

First, set the USB output interface resolution to 24b. This is important because mic signals in a quiet room (35dBA) may be quantised away in a 16b audio capture. Also, 24b audio has been found to work on most hosts.

Second, configure the audio crossbar to output `PACKED_ALL` on USB output channels 0 and 1.

This can be done by setting the parameters in the data partition. To do this, navigate to the `data-partition` directory of the Release package. Note, it recommended to make a copy of the default .json config file for future reference.

To create the “packed output to USB” commands in file `input/set_packed_all_on_usb_out.txt` Add the following contents to this file.

```
SET_IO_MAP 0 16
SET_IO_MAP 1 16
```

Note that the IO map source 16 is set for both USB output channels. Source 16 automatically resolves the channel indices so this will result in a stereo output containing a packed capture of all six discrete channels of interest.

Next, add the following sections to a `ua_24b_packed.json` file item section and save it:

```
        {
            "path": "input/set_packed_all_on_usb_out.txt",
            "comment": ""
        },
        {
            "path": "input/device_to_usb_bit_res_24.txt",
            "comment": ""
        },
```

Now we generate the data partition from our updated json file:

```
 python3 xvf3510_data_partition_generator.py ua_24b_packed.json
```

This will generate the new data partition file as follows:`output/data_partition_factory_ua_24b_packed_v4_0_0.bin`.

Finally `cd` up one level to root of the release package and flash the firmware along with the newly created data partition configuration file:

```
xflash --no-compression --boot-partition-size 1048576  bin/app_xvf3510_ua_v4_0_0.xe
\
--data data-partition/output/data_partition_factory_ua_24b_packed_v4_0_0.bin
```

Once the firmware has booted following the flashing operation it can be verified in the sound control panel that the USB input stream from the XVF3510-UA to the host is now set to 24b.

Next the audio of interest is captured. Do this with a wav capture utility to capture the stereo output from the USB input from the XVF3510 device at 48kHz. Ensure the file is saved as 32b Signed Integer which is needed for the next step. Note that viewing/listening to the packed wav is non-sensical because it contains packed/multiplexed signals and will sound noisy.

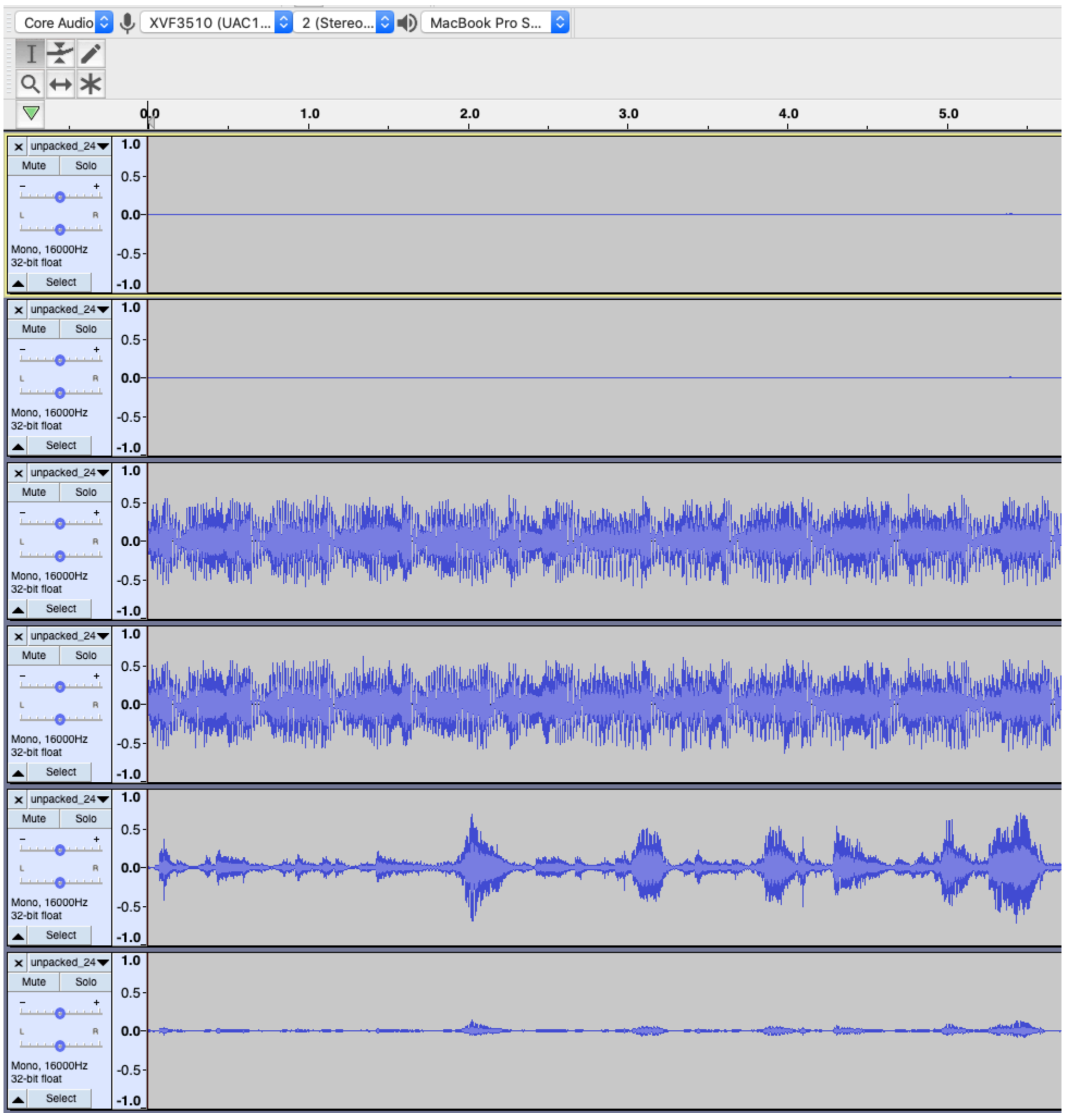
Finally convert these packed files into unpacked, 16kHz, 6-Channel audio files.

```
python host/unpacker_packed_all.py packed_capture.wav unpacked_6ch_16kHz.wav 24
```

The output file `unpacked_6ch_16kHz.wav` may now be inspected. Note that the channel assignment is as follows:

- Microphone Ch 0
- Microphone Ch 1
- Reference input Left
- Reference input Right
- Pipeline Output Ch 0 (nominally ASR)
- Pipeline Output Ch 1 (nominally Comms)

Below is a visualisation of a six-channel audio capture. Note the relatively quiet mic signals compared with the reference. This is typical and allows for loud near-end signals without distortion.



PACKING SPECIFIC SIGNALS

`PACKED_PIPELINE_OUTPUT`, `PACKED_MIC`, `PACKED_REF` all use the same underlying packing function. They pack 2 channels (mic0/1 or ref/ref_r or pipe0/pipe1) into a single audio channel. **It requires that the output interface, including host processing, be capable of bit-perfect 32b audio**. It packs the two 16kHz samples into three 48kHz samples as follows:

- top 24b of sample[0] with 8b LSB marker '0x00'
- top 24b of sample[1] with 8b LSB marker '0x01'
- the bottom 8b of sample[0], the bottom 8b of sample[1], 0x00, 8b LSB marker '0x02'

The `unpacker.py` script then looks for 0x00, 0x01, 0x02 in the LSByte to check for a packed sequence. So inspecting the wav in a hex editor should make it clear when it is captured properly.

It will capture bit-perfect data.

Packing specific signals will not work on a Mac because it only supports 24b audio due to core audio representing audio using single-precision floating-point. It has been tested and works well on Linux (x86/RPI) which supports bit-perfect 32b audio.